Session 2018-19 (Even Semester)

Eighth Semester **Subject: Gaming Architecture & Programming**

NOTES

1. (a) Write a program in C++ to capture the mouse and draw lines. [10 Marks]

1.Monitor for WM_LBUTTONDOWN.
2. Capture the mouse using SetCapture().

3. Save the position of the mouse relative to your window; this is usually captured in a static variable like static POINT startingPoint={0};This is the starting point of the line.
4. Make sure you have the image information ready to re-paint on every WM_MOUSEMOVE.
5. Set a boolean flag indicating the user has started to draw a line. The simplest form of this flag is static boolbDrawingLine=false;

6. Monitor for WM_MOUSEMOVE. If the boolean flag is set, draw the contents of the window, then draw the potential new line.
7. Monitor for WM_LBUTTONUP. If the boolean flag is set, draw the contents of the window, then draw the now permanent line.

Drawing Lines with the Mouse-

When the window procedure receives a WM_LBUTTONDOWN message, it captures the mouse and saves the coordinates of the cursor, using the coordinates as the starting point of the line. It also uses the clipcursor function to confine the cursor to the client area during the line drawing operation.
During the first WM_MOUSEMOVE message, the window procedure draws a line from the starting point to the current position of the cursor. During subsequent WM_MOUSEMOVE messages, the window procedure erases the previous line by drawing over it with an inverted pen color. Then it draws a new line from the starting point to the new position of the cursor.
The WM_LBUTTONUP message signals the end of the drawing operation. The window procedure releases the mouse capture and frees the mouse from the client area.

b) What are sprites and why are they used? Write a basic sprite class and explain the various properties in it? [10 Marks]

CSS Sprites are a means of combining multiple images into a single image file for use on a website, to help with performance.

Sprite may seem like a bit of a misnomer considering that you're create a large image as opposed to working with many small ones, but the history of sprites, dating back to 1975, should help clear things up.

To summarize: the term "sprites" comes from a technique in computer graphics, most often used in video games. The idea was that the computer could fetch a graphic into memory, and then only display parts of that image at a time, which was faster than having to continually fetch new images. The sprite was the big combined graphic.

CSS Sprites is pretty much the exact same theory: get the image once, and shift it around and only display parts of it. This reduces the overhead of having to fetch multiple images.

It may seem counterintuitive to cram smaller images into a larger image. Wouldn't larger images take longer to load?

Let's look at some numbers on an actual example:

Image	File Size	Dimensions
canada.png	1.95KB	256 x 128
usa.png	3.74KB	256 x 135
mexico.png	8.69KB	256 x 147

That adds up to a total of 14.38KB to load the three images. Putting the three images into a single file weighs in at 16.1KB. The sprite ends up being 1.72KB larger than the three separate images. This isn't a big difference, but there needs to be a good reason to accept this larger file... and there is!

While the total image size (sometimes) goes up with sprites, several images are loaded with a single HTTP request. Browsers limit the number of concurrent requests a site can make and HTTP requests require a bit of handshaking.

Thus, sprites are important for the same reasons that minifying and concatenating CSS and JavaScript are important.

2. a) What are the phases in Game-play development? Explain the process, people involved in each phase? [10 Marks]

- Step One: Initial Planning

The Genesis Gaming Design and Marketing teams will meet with the client to determine the key concepts driving the development of a strategic game portfolio. This will address issues such as analysis of an existing portfolio, current demographics, additional player acquisition and retention, emerging trends and profiles. We will further discuss a range of themes, volatility levels, features, bonus games and any other aspect required for bespoke development of a strategic portfolio. It is also important that there is an overall marketing discussion to determine how the games can be used to further extend the client's brand.

- Step Two: Technical Review

Our Engineering Team will review documentation provided by the client to determine if there are any technical questions or matters that need to be addressed to best develop to a specific platform.This will ultimately save significant time in the integration process.

- Step Three: Initial Themes & Concept Art

We will submit themes, names, concept art and descriptions for consideration and approval or modification. We will then work with the client to determine portfolio development priorities. This allows the ability to set expectations as to game delivery. The client can then implement a schedule for marketing and release.

- Step Four: Features and Mathematics

Our Game Design and Mathematics departments will determine and create unique math models that best represent the client's objectives as stated through the analytical process and the client's objectives. Math will be verified and all required percentages will be provided.

- Step Five: Art and Creative Design

The Genesis Gaming Art Department will design and submit static art and design elements for approval. Subsequently, our Animation and Music Composition teams will finalize the game. This will require the client to provide appropriate documentation such as templates and other substantive game specifications for consistency.

Upon completion, our Demo team will provide a playable version of the game for additional review.

- Step Six: Integration

Upon approval, our Engineering Department will integrate the game to the appropriate provider platform. This requires the necessary documentation, such as an API, from the client along with server support, assuming it

is not an "asset only" delivery. We place a significant emphasis on product assurance and quality control so the game will be very "client provider friendly."

- Step Seven: Delivery

All game assets, in the requested file formats, including all release documentation will be provided according to contractually specified delivery requirements. Genesis Gaming is also happy to provide any material that may be requested for the client's marketing campaign.Since success is our mutual objective, it is our goal to make certain that the client has all tools possible to promote the best exclusive content in the industry

b) Describe Blue-Sky research in detail. [10 Marks]

We would all love to spend all day in front of a computer munching pizza and guzzling coffee, freely engaging our creativity and researching whatever we like. Some companies do allow this and have active (if somewhat variable) research departments. That's a good thing. Research is essential to the survival of a company. It's *blue-sky* research that you have to be careful about. By definition, any technology present in games currently available in the shops is about 18 months behind what is currently being worked on by the best design houses.

In other words, even with the best will in the world, it is statistically unlikely that anything remotely useful will be achieved, except a rather tragic comedy of errors. When a game project depends on the outcome of research that has not been completed, that project is in great danger. Putting anything in the critical path for which the outcome cannot be predicted is sheer idiocy; but for some reason—whether greed, stupidity, or just plain ignorance—development teams seem to do this on a regular basis.

We're not advocating that research should be abolished; that would be a draconian measure and would lead to further stagnation of the industry. What we *are* saying is that there is a time for research and a time for development, and that the two should never overlap. Any research that is instigated should be directed research. It should have an aim.

An example would be the design and development of a library useful in fuzzy-logic calculations. For sure, there would be a fair amount of research involved, but this would be research into ways of optimizing and improving on known techniques. The blue-sky alternative would be to look for a completely new method of doing things. Fortunately, there is very little you can imagine doing with a computer that hasn't been already done by someone somewhere.

If you're very lucky, then they have published their results. Building on the work of others, although less glamorous, is a surer way of getting good results. Besides, if you wanted glamour then why did you become a

developer? When your team is researching, set a strict time limit and stick to it ruthlessly. Run with what you physically have at the end of the research period, not with what you are promised at the beginning.

The time allocated for research should also factor in the time necessary to bring the research project to a successful conclusion and with a stable and well-documented component. As the research progresses, more and more unknowns will become quantifiable. As soon as the fundamental technique being researched is working, a mini-schedule can be drawn up to allow for this tidying-up procedure. If the game design depends on this research, then this is the single most critical point of the whole development.

The project will succeed or fail—right here, right now—dependent on the success of this research project. Obviously, this is not a good idea: Gambling the whole project on blue-sky research is a game that only the very foolhardy or the extremely desperate would play. Remember, any safety net is better than no safety net. If no safety net is possible, then "make-or-break" research should be avoided. It's too much of a risk, no matter how cool the developer doing the research is.

Under most circumstances, releasing the product is better than canning it. Only the most cash-rich companies will be able to afford internal research such as that used during the development processes of *Die by the Sword* and *Outcast.* Not so many companies are able to afford the protracted development times and the risks that are associated with research of this nature. If your company doesn't have the sort of cash available to finance a research department, you still have a few options open.

The first, and most undesirable, is to avoid projects that require R&D. This is only really acceptable for the "ticking-over-and-we-know-we'll-only-sell-a-couple-of hundred- thousand-or-so" type of company. This is a viable option, but it isn't going to go anywhere fast.

For the less-affluent companies, another option is to form links with academia. Forming a loose relationship with universities and their computing departments can provide some good technology at some very good prices. A company that we have worked for in Belgium used their local university as a source of cutting-edge cryptography algorithms and regularly tested their latest algorithm designs by letting the university researchers loose on them. The theory was that any implementations of algorithms that these university cryptography experts couldn't crack were pretty much guaranteed to be secure.

If you use the "external research" system, then this also simplifies some other company decisions. Form a liaison with academia: A good relationship means that they get to publish and you get to use their new ideas. Yet another option is to be part of a large conglomerate, either using the Hollywood studio model or the loose alliance model typified by Lionhead's satellite. The latter may not be an optimal system.

To optimize it you need to ensure that the separate development teams are aware of and able to share each other's technology. Also you would ideally organize collateral R&D so that two companies don't waste time

on the same tasks, all of which is best achieved by appointing a resource investigation unit to interface between all the projects in development. A loose development alliance can work, therefore, but it needs structure.

c) Define middleware. Describe the popular 3D engines currently in use. [10 Marks]

Middleware has two separate but related meanings. One is software that enables two separate programs to interact with each other. Another is a software layer inside a single application that allows different aspects of the program to work together.

The most common type of middleware is software that enables two separate programs to communicate and share data. An example is software on a Web server that enables the HTTP server to interact with scripting engines like PHP or ASP when processing webpage data. Middleware also enables the Web server to access data from a database when loading content for a webpage. In each of these instances, the middleware runs quietly in the background, but serves as an important "glue" between the server applications.

Middleware also helps different applications communicate over a computer network. It enables different protocols to work together by translating the information that is passed from one system to another. This type of middleware may be installed as a "Services-Oriented Architecture" (SOA) component on each system on the network. When data is sent between these systems, it is first processed by the middleware component, then output in a standard format that each system can understand.

Middleware can also exist within a single application. For example, many 3D games use a "3D engine" that processes the polygons, textures, lighting, shading, and special effects in the game. 3D engines are considered middleware, since they bring different aspects of the game together.

Unreal Engine 4 is a complete suite of game development tools made by game developers, for game developers. From 2D mobile games to console blockbusters, Unreal Engine 4 gives you everything you need to start, ship, grow and stand out from the crowd.

Revolutionary new workflow features and a deep toolset empower developers to quickly iterate on ideas and see immediate results, while complete C++ source code access brings the experience to a whole new level. Unreal Engine technology powers hundreds of games as well as real-time 3D films, training simulations, visualizations and more. Over the past 15 years, thousands of individuals and teams and have built careers and companies around skills developed using the engine.

UNREAL ENGINE 3- Unreal Engine 3 is the complete toolset to create your own games. It is a very widely used game engine in the industry. Unreal Engine is very versatile and has been used to create many triple A games such as: Batman: Arkham City, Gears of War Series, Borderlands Series. Unreal Engine is one of my favorite engines to work with. UDK is a free educational version of the engine, with commercial license available if you want to take your project further and sell it.

CRY-ENGINE- CryEngine 3 has been used in games such as Crysis 2, Crysis 3 and Sniper: Ghost Warrior 2. Just like Unreal Engine 3, CryEngine 3 is the complete toolset for game development. CryEngine 3 has been used as a benchmark for visual graphics for some time and it continues to push the limit what games are capable of. One of CryEngine's features is its ability to produce huge beautiful, highly detailed landscapes. CryEngine 3 SDK is now on Steam and requires a monthly subscription service. You can also choose a full license but for independent game developers and hobbyist, Steam subscription will be enough.

SOURCE ENGINE- Source Engine has been used very extensively in the modding community with hundreds if not thousands of mods available. Source Engine is a bit outdated, yet still very powerful. It has been used to create games such as Half Life 2 series, Counter-Strike: Source, Counter-Strike: Global Offensive, Left4Dead, Left4Dead 2, Portal 1 and 2. Each game has a huge community behind it with new content always being released. I love using Source Engine because of its games. You can get your hands on the engine by downloading any of Valve's released games on Steam. If you are interested in licensing Source for commercial project go here.

UNITY 3D- Unity3D has been a very popular choice among developers. Full pledge game engine featuring everything you would need to create full 3d or 2d games with multi-platform support right out of the box. Easy engine to get into and begin using. Unity has a free indie version as well as commercial license version. Latest version of Unity offers DirectX 11 support.

TORQUE 3D-Originally Torque was developed for 2001 FPS shooter, Tribes 2. Torque is an open source game engine and has been an independent dev favorite for quiet some time. Some features include a world editor, Collada support, per pixel dynamic lighting, normal and parallax occlusion mapping, reflections, sky system, physX, multiple platform publishing and access to source code.

BLENDER- Blender is a free and open-source 3D content creation suite. It includes tools for animation, compositing, 3D modeling, uv unwrapping, texturing, rigging and skinning, fluid and smoke, particle system, physics and rendering. It also has a built-in game engine. The game engine is written in C++ and includes support for Python scripting and OpenAL 3D sound. I like Blender because it is the only free alternative that I know of to Maya/3dsMax as a modeling/animation software.

NEOAXIS- Neoaxis has all the features of a modern engine such as advanced material and shading support, real-time shadows, built-in Nvidia physX and current/next-gen rendering. It comes with complete pipeline SDK, including a map editor. When you download the free educational version of the engine it comes with example files for first person shooter with multiplayer support, real-time strategy and 3rd person shooter. There is a free non-commercial SDK available to download.

3. (a) Explain in detail Cleanup process to be followed during and after the game exit. [10 Marks]

(b) What are tokens? Explain tokenization in pong game specifying interaction matrix and the sequence of events that occur when a goal is scored. [10 Marks]

In general, a token is an object that represents something else, such as another object (either physical or virtual), or an abstract concept as, for example, a gift is sometimes referred to as a token of the. We can also consider these tokens to be arranged in a form of hierarchical structure.

The playing area, or game world, in itself is at the top of the hierarchy. From then on in, it is an essentially flat hierarchy.

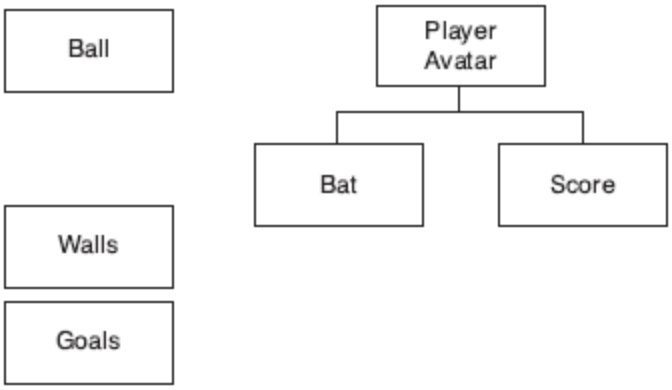


Figure :The *Pong* token hierarchy.

The game world token contains all the other tokens. Obviously every token has to operate within the game world in order to form a part of the game. The player avatar token is the representation of the player within the game world. It is effectively a channel for the user interface between the player and the game.

The player avatar for *Pong* is very simple; it is merely a bat and a score. These are how the player is represented in *Pong*. The other tokens—those manipulated by the computer—are the ball, the walls, and the goal zones.

Now it's time for a little sleight of hand of the sort that is possible with only the written word. Reread the two paragraphs, and for every instance of the word "token," read it as "object."

So, if we were just talking about objects all along, why didn't we just use the word "object" to start with?

The main reason—and why we particularly like the use of the word token and why we will continue using it from here on in—is that these conceptual tokens may not have a one-to-one mapping with the programming language objects (for example, in C++) that are defined by the programmers. What we are trying to do is to break down the game design into conceptual objects that will eventually be translated into programming language objects. This tokenization process is an intermediary stage in the production of a decent architecture. In order to describe this without causing confusion, we need to use different terminology for each type of object.

The tokenization of a game design such as *Pong* is fairly trivial, and there's really only one way to do it. In spite of this, it makes an excellent example to try to demonstrate the thought processes behind tokenization. Not all games will be so trivial, and, for some more complex games, there may be many ways, all of which are equally valid. So now we have a set of tokens. On their own, they are not very exciting as they do not interact with one another. But as we know, in *Pong* there are all sorts of interactions going on. Well, one anyway: collisions.

We can now define an event—the collision event. Let's say that a collision event is generated when two tokens collide. The net result of this event is that each token receives a message telling it that a collision has occurred, and the type of object it has collided with.

The token interaction matrix is a very important construct. It is a chart of all the interactions that take place in the game. Note that for very large games we would not use a token-token matrix directly. Instead we would introduce an extra layer of abstraction by using token-property and/or property-property.

Okay, so let's look at the *Pong* token interaction matrix. The matrix is arranged in a triangular format, with each token listed along the side and the bottom. An unusual feature of *Pong* is that tokens do not come into contact with other tokens of the same type. This immediately means that the token-token interactions for bat-bat, ball-ball, wall-wall, goal-goal, and score-score can be discounted. Due to the nature of the game, the following interactions can also be discounted: bat-goal, wall-goal, score-bat, score-ball, and score-wall.

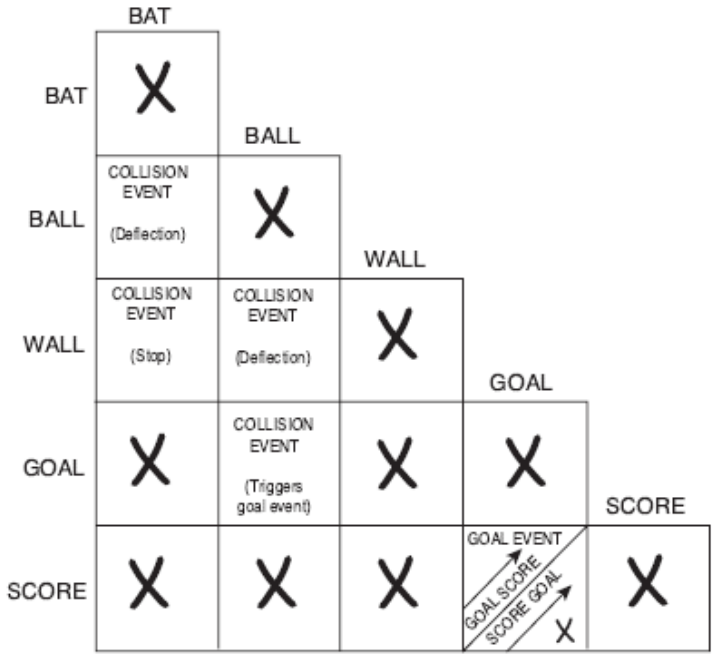


Figure : The *Pong* token interaction matrix.

4. (a) State the design patterns that are commonly used in game design and explain any four with examples. [10 Marks]

Design patterns are generalized solutions to generalized problems that occur with some modicum of frequency when you're creating software using the object oriented programming paradigm. Why Use Design Patterns? The most basic and condescending answer is: because these solutions have existed for a relatively long time and many experts have used them, they're likely better than any solution you could come up with on your own.

Common Game Programming Patterns

Singleton - You create objects that ensure that only a single instance of which can exist at a time. In my game Total Toads, this is the design pattern used because it was easiest to fit with cocos2d's design. For example, in cocos2d, there's a Singleton CCDirector, CCSpriteFrameCache, etc. It seems this is an often-used panacea in game programming.

Though the general consensus seems to be that it isn't a panacea so much as it is a cancer because it actually masks poorly designed architecture. You should probably avoid using this design pattern if you can

because there's likely a better way to design the architecture of your game. This can avoid the problem of having multiple instances of objects of which there should only be one, like a "Player" object in a single player game.

Factory - You create an object whose purpose it is to create other objects. For example, you can have a factory class called "GameObjectFactory" with static (possibly parameterized) methods to create other game objects like a "Player", "Enemy", "Gun", or "Bullet". The latter classes might have complex constructions that make obtaining a instance of that specific class difficult.

The factory can take care of the object's complex configuration (adding to an object pool, adding to physics engine, etc) and simply return a reference to the created object. This pattern helps you avoid the problem of complex object instantiation by keeping these complex configurations in a single place, rather than scattered around in your code.

Observer - The object in question maintains a list of other objects that are interested in its state and notifies these listening objects of a change in its state. In Total Toads, we have 3 Frog objects and 3 FrogAnimation objects that can control the animation of the frogs based on their state. In this case, the FrogAnimation objects are observers of the Frog objects.

Every time the state variable for a Frog changes, it notifies the associated FrogAnimation object to notice the new state and take an action if necessary (like animating the frog). This pattern helps you avoid the problem of event notification in your game. Since games are user-interaction driven, objects can change state at almost any time. When an object changes state oftentimes that object needs to be animated or have other objects change their state with respect to the new state.

State - You have an abstract (empty implementation) class that has subclasses to define the current state. An example of this might be a first person shooter that has a "Player" class who has several possible states like "PlayerInCombat", "PlayerOutOfCombat", and "PlayerInMenu".

When the state is changed, the player shall be represented as an instance of the appropriate state class. When the player starts to be shot at, the player object "switches" to an instance of the PlayerInCombat class to take advantage of that class's implementation of the mouse's left button click which makes the player able to shoot their gun. Similarly for the "PlayerOutOfCombat" and "PlayerInMenu" classes, where the mouse might allow usage of items or the clicking of menu options, respectively.

This pattern helps you avoid monolithic methods that perform differently based on the object's state by having a bunch of switch or if/elses in your code. Instead, the object just changes and runs its appropriate code. This also simplifies your code and allows you to more easily find problems in your objects behavior since the state is right in the object's class name.

(b) Discuss the seven golden principles of effective game design. [5 Marks]

Gambit 1 –Reuse

The transition of any cottage industry to a major industry comes when you achieve reuse. In this it is even more useful is the foundation classes that can be carried over from game to game. The design patterns, whose value is that they enable a design to be built rapidly and in a form that is already largely understood at the super modular level. Developers often disdain plug-in modules because of the "Not Built Here" syndrome.

Frequently, even a perfectly good in-house module is not used merely because it was created by a different team in the same company. At the very least, plug-ins should be used in the early stages. If you genuinely feel there's a better route finding algorithm, you can always write it later. Reusable components require a standardized architecture, and you need to know the format that each module's inputs and outputs will take. Then, you can refine the interior of the module as a black box without having to change the rest of the system.

Gambit 2- Documentation

Once we talked to a development manager who was lamenting the loss of a project that had been running smoothly for nine months. "It was a dream," he said. "It was on schedule, they had a good demo, the whole team got on, and the concept was a winner, too." "What went wrong? Let me guess—the publisher just didn't get it? "No, the publisher was behind it all the way! The trouble was the lead programmer went off to a new job in Atlanta, and nobody else can complete his engine."Documentation won't always save you from situations like that one.

Documentation aims to make individuals less dispensable, but it can't always make them indispensable. However, look at it like an airbag in your car: it's not guaranteed to save your life, but it sure helps with the odds. Documentation has another great benefit also. It allows other team members to know what your work is about. The payoff to the whole team is that developers don't have to keep interrupting each other's work with minor questions: they can go to the documentation. And it isn't just for other team members, either: documentation often reminds you of a train of thought you might have forgotten.

Gambit 3- Design First

Design comes first, and development comes second. Be careful with that dictum. It doesn't mean that design stops and then development begins. Design is an ongoing process, which we've estimated is roughly 80% complete at the commencement of development and which is further refined as time goes on.

"Wrapping Up," implies that approximately 20 man-months went into the original Populous game. Technology has moved on since then, forcing games to get bigger, and it's possible that the latest Populous game took more than 200 man-months. However, there would be nothing to stop the first 10% of that time being spent on a throwaway test-bed. The game play of Populous 3 is not so far advanced beyond the original, and test-beds are very easy to create.

Gambit 4- Schedule

Sticking to the schedule is all important.

Let's hammer the point home.

Teams make plans and then routinely abandon them when they run into schedule trouble. The problem isn't so much in abandoning the plan as in failing to create a substitute, and then falling into code-and-fix instead.

Just like plans, you can't always stick to your first schedule, but you can try to recalibrate the schedule as you go on. Suppose you had to get to an interview at noon and in the morning you're planning to get your haircut and buy a new suit. Only you leave the house late. What do you do? Just press on with all your tasks and hope you still make the meeting at noon? Of course not!

Yet this "hope to catch up" approach is common in development teams. Instead, you re-plan the schedule, trimming nonessential tasks or finding ways around them. In this example, maybe you'd find a hairdresser nearer to the tailor's—not your favorite hairdresser, but it'll have to do. Or maybe you'd try to reschedule the interview. The point is that you would hopefully realize that the worst thing to do would be to run around aimlessly without any kind of schedule.

Gambit 5- Catch Mistakes as You Go Along

The longer an error is allowed to fester untouched, the higher the cost of repair (in terms of both time and money) when it is finally tackled. This could make the difference between a successful project and a cancellation.

Gambit 6- Limit R&D

R&D is effectively open ended and virtually impossible to schedule for, especially if you are dependent on the results. The rule is to perform R&D outside the scope of the project and to make sure that the success of the project is not dependent on the outcome. Always have a backup plan.

Gambit 7- Know When to Draw the Line

When is it "good enough"? The average game developer will say "Never! Or at least not until I've optimized every last function down the max." Sometimes developers are just too focused to know when to stop

and look at the bigger picture. This is a very difficult skill for a developer, who is generally used to focusing in at a very detailed level of construction. A project planner is the individual responsible for keeping the developers focused in the right areas and for the right periods, but this role can fall to other team members, too. Knowing when to draw the line prevents feature creep, one of the three lead balloons that can affect a project.

(c) Give a practical example of :- [5 Marks]

i. Using Inheritance over Containment.

ii. Using Containment over Inheritance

(d) Describe the game build process. [5 Marks]

Depending on the type of game you are creating and on how you expect to get it produced, you may want to develop as many as six different documents at various stages of the creative process. If it sounds like a lot of work, then you are beginning to get my point

- All games should begin with a design treatment, i.e., a quick discussion of your product's unique features and target audience.
- Then, you should move to a preliminary design, discussing the game's rules, content and behavior in a purely qualitative way. This document should be circulated and discussed as widely as possible given the situation.
- A final design is a re-write of the previous document, which etches the product's features in stone.
- The product specification (which only really makes sense for interactive products) details how the features adopted in the final design will be implemented.
- The graphic bible determines the look and feel of the game's characters, maps, props, etc.
- The interactive screenplay, if appropriate, contains the dialogs and the storyline implemented into the product.

The game designer may or may not be qualified to write all of these documents. A product specification for a computer game, for example, requires considerable input on the part of the game's producer, lead programmer and lead artist, while a screenplay should be crafted by a professional writer. However, as "guardian of the vision", the designer should have final say on what goes into his product, and be involved in all aspects of the process, if only as an overseer.

5. (a) Explain why game development has to be tier-based? Describe the application of Tier-Based approach to architecture design. [10 Marks]

Tier Zero: The Prototype-

Of course, it would be difficult to leap straight in and implement the skeleton correctly the first time you try, so we need to take this into account when designing the architecture. We do this via prototyping techniques. In fact, the prototype is really the very first part of defining the architecture. We could say that the prototype is Tier Zero in the development model. Tier Zero is really a special case, before we get into the architecture proper. We need to be able to test our game design ideas, refine them, and work out what we need to do in order to efficiently implement our tier-based architecture. This tier of our project isn't really one of the main tiers.

Tier One and Beyond

Once all the prototyping has produced useful results, it's time to get down to the real meat of the architecture design. Tier One of the development process takes the results of the prototyping we have already done and unifies the results into a single framework design. The good thing about this is that this framework will be designed to be reusable. If we are developing our code as reusable components, then a large proportion of this initial Tier One work will be useful for other concurrent or future projects. In Tier One, the developers concentrate their efforts on producing the hard-architecture components that will be required in the game project. Usually, unless there is a pressing reason otherwise, Tier One begins at the lower granularity levels, and the work involves producing components such as hardware-interfacing modules and the basic game framework code.

N-tier and 3-tier are architectural deployment styles that describe the separation of functionality into segments in much the same way as the layered style, but with each segment being a tier that can be located on a physically separate computer. They evolved through the component-oriented approach, generally using platform specific methods for communication instead of a message-based approach.

N-tier application architecture is characterized by the functional decomposition of applications, service components, and their distributed deployment, providing improved scalability, availability, manageability, and resource utilization. Each tier is completely independent from all other tiers, except for those immediately above and below it. The nth tier only has to know how to handle a request from the n+1th tier, how to forward that request on to the n-1th tier (if there is one), and how to handle the results of the request. Communication between tiers is typically asynchronous in order to support better scalability.

N-tier architectures usually have at least three separate logical parts, each located on a separate physical server. Each part is responsible for specific functionality. When using a layered design approach, a layer is deployed on a tier if more than one service or application is dependent on the functionality exposed by the layer.

An example of the N-tier/3-tier architectural style is a typical financial Web application where security is important. The business layer must be deployed behind a firewall, which forces the deployment of the presentation layer on a separate tier in the perimeter network. Another example is a typical rich client connected

application, where the presentation layer is deployed on client machines and the business layer and data access layer are deployed on one or more server tiers.
The main benefits of the N-tier/3-tier architectural style are:

- Maintainability. Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.
- Scalability. Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.
- Flexibility. Because each tier can be managed or scaled independently, flexibility is increased.
- Availability. Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Consider either the N-tier or the 3-tier architectural style if the processing requirements of the layers in the application differ such that processing in one layer could absorb sufficient resources to slow the processing in other layers, or if the security requirements of the layers in the application differ. For example, the presentation layer should not store sensitive data, while this may be stored in the business and data layers. The N-tier or the 3-tier architectural style is also appropriate if you want to be able to share business logic between applications, and you have sufficient hardware to allocate the required number of servers to each tier.
Consider using just three tiers if you are developing an intranet application where all servers are located within the private network; or an Internet application where security requirements do not restrict the deployment of business logic on the public facing Web or application server. Consider using more than three tiers if security requirements dictate that business logic cannot be deployed to the perimeter network, or the application makes heavy use of resources and you want to offload that functionality to another server.

(b) What is source control? Explain in brief the different functionalities provided by Source Control System? [10 Marks]

A component of software configuration management, version control, also known as revision control or source control,[1]:2 is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number," "revision level," or simply "revision." For example, an initial set of files is "revision 1." When the first change is made, the resulting set is "revision 2," and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated, when the era of computing began.

The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may change the same files.

Version control systems (VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, e.g., Google Docs and Sheets[2] and in various content management systems, e.g., Wikipedia's Page history. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming.

Software tools for revision control are essential for the organization of multi-developer projects.

In computer software engineering, revision control is any kind of practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code.

As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops).

Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently (for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used on many large software projects. While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of self-discipline on the part of developers, and often leads to mistakes. Consequently, systems to automate some or all of the revision control process have been developed.

Moreover, in software development, legal and business practice and other environments, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary interests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even indispensable in such situations.

Structure revision control manages changes to a set of data over time. These changes can be structured in various ways.

Often the data is thought of as a collection of many individual items, such as files or documents, and changes to individual files are tracked. This accords with intuitions about separate files, but causes problems when identity changes, such as during renaming, splitting, or merging of files. Accordingly, some systems, such as git, instead consider changes to the data as a whole, which is less intuitive for simple changes, but simplifies more complex changes.

When data that is under revision control is modified, after being retrieved by checking out, this is not in general immediately reflected in the revision control system (in the repository), but must instead be checked in or committed. A copy outside revision control is known as a "working copy". As a simple example, when editing a computer file, the data stored in memory by the editing program is the working copy, which is committed by saving.

Concretely, one may print out a document, edit it by hand, and only later manually input the changes into a computer and save it. For source code control, the working copy is instead a copy of all files in a particular revision, generally stored locally on the developer's computer in this case saving the file only changes the working copy, and checking into the repository is a separate step.

If multiple people are working on a single data set or document, they are implicitly creating branches of the data (in their working copies), and thus issues of merging arise, as discussed below. For simple collaborative document editing, this can be prevented by using file locking or simply avoiding working on the same document that someone else is working on.

Revision control systems are often centralized, with a single authoritative data store, the repository, and check-outs and check-ins done with reference to this central repository. Alternatively, in distributed revision control, no single repository is authoritative, and data can be checked out and checked into any repository. When checking into a different repository, this is interpreted as a merge or patch.

6. (a) Explain the various platforms on which game can be deployed on? What are the advantages and disadvantages of each of these platforms? [10 Marks]

Finding the right game engine can be the key to successfully building and deploying a game that becomes both popular and lucrative. But there are so many game engines out there vying for your attention. Clearly, some guidelines on the subject would be useful.

Ten years ago, it was okay to release your game on one platform at a time. Today, it’s more typical for a game to be released rapidly on multiple platforms. To that end, a cross-platform game engine offers some real advantages, and the options there are quite diverse and plentiful. Having recently released my own game template based on

Cocos2D JS, I thought it would be interesting to compare some of the major game engines and see how they stack up against each other.

To prepare for this post, I wrote a complete Breakout clone in four of today's top cross-platform game engines: Unity, Corona, Cocos2D JS and Appcelerator Titanium, and also using my game template, RapidGame Pro. The code can be found at the end of each section, so you can see for yourself. My observations on how they all compare should help you make a choice that may save you and your team weeks or months.

Unity : Unity is, in short, a closed-source, cross-platform game development application. You create your game by manipulating objects in 3D and attaching various components to them. Even 2D games must be manipulated in 3D. Scripts are written in C# (recommended), Boo or Unityscript (mistakenly called JavaScript) and attached to 3D objects as components.

Launching Unity for the first time, you may feel like the pilot of a 747 jet plane. There is much to learn before even the first switch can be flipped. First of all, there's camera and lights. When trying to add a simple cube to the scene, it can get lost behind the camera or perhaps be invisible because there's no light. In short, there is a learning curve.

Prepare to spend approximately 8-12 hours getting familiar enough to develop your own game. Unity was first released in 2005 and the interface hasn't changed much since. To be frank, it feels like many of the repetitive tasks in day-to-day Unity game development are busy work. Adding audio sources, updating prefabs and importing assets are all examples of tasks that shouldn't have to be done, or shouldn't take so long. It would be nice if Unity had a modern makeover.

That said, once you've created a game with Unity, deployment is a cinch. With a couple of clicks, you can export your game to mobile, desktop and/or web (web currently requires the Unity player app to be installed). If you have the right license, you can even deploy to gaming consoles like Xbox, Playstation and Wii.

Corona : Corona is a closed-source 2D game simulator and cloud-build application. Game code is written in Lua scripts and played back in the Corona simulator. Like Mystique from X-Men, the simulator can take on many skins, resolutions and ratios. When you're ready to deploy, it builds your game in the cloud and delivers you an iOS or Android game client.

One shortcoming of Corona is its limited deployment options. Only mobile platforms like iOS, Android, Kindle and Nook are supported. Windows Phone is coming soon. Cloud-Imagine a day full of testing your game on the device, tweaking one little thing and waiting a few minutes to be able to see if it worked.

Like Unity, Corona is closed-source and proprietary. There's no way to make a modification or fix a bug in the engine, and you cannot learn from its code.

Cocos2D JS : Cocos2D JS is a cross-platform, open-source, free game development SDK. It is the newest — and perhaps sexiest — member of the Cocos2D family. Essentially it's a combination of two popular open-source projects: Cocos2D X for mobile / desktop and Cocos2D HTML5 for web. While it is currently 2D / 2.5D, there are plans to add 3D support.

You write game code entirely in JavaScript. On native platforms like mobile and desktop, your game's JavaScript is bound to native C++ objects, granting you maximum speed without having to write any native code. Web platforms run pure JavaScript and render using Canvas or WebGL, so no player applications need to be installed.

The easiest way to get started with Cocos2D JS game development is using the HTML5 platform. Open up a browser window and your favorite text or code editor, save your JavaScript, refresh the browser and voila. It's a rapid way to develop. When you're ready to test and deploy to native platforms, you'll need Xcode, Visual Studio and/or Eclipse.

Cocos2D JS games can currently be deployed to iOS, Android, Blackberry, Windows Phone, Mac, Windows, Linux and HTML. With such wide deployment options, it's easy to see why many game developers are choosing Cocos2D.

Appcelerator Titanium : Titanium is a cross-platform, open-source app development kit and Eclipse-based IDE. Apps are written in JavaScript and run natively, not just in a WebView. With Titanium Studio it's possible to develop, test and deploy to mobile and web platforms.

For 2D game development, there's the Platino Game Engine, an open-source — but not free — SDK that can be added to your Titanium stack. Getting acquainted with Titanium (more specifically Platino) is not as easy as it could be. The documentation has holes. For example, the crucial .center sprite property is left undocumented. Moreover, the physics engine is cumbersome and archaic. You have to synchronize all physics bodies and sprites manually using a very non-JavaScript, C-like API.

On the bright side, one nice thing about Titanium development is that the SDK is prebuilt. You can run your game on a simulator or device with very short build times.

RapidGame Pro : RapidGame Pro, an open-core game (dual MIT licensed) template based on Cocos2D JS, to make game development using open source more rapid. It achieves this in a few ways:

By providing a project creator tool and game templates that make starting a game with scenes, sprites, sound, physics, a server, monetization, social, etc. a breeze.By prebuilding native libraries.

By providing and incorporating plugins for IAP, displaying ads, social networking,analytics,asynchronous multiplayer and virtual economies that work on all platforms. By including example code to a complete game based on multiple currencies.

Some of RapidGame Pro's plugins had to be developed from scratch for multiple platforms. For example, the Facebook plugin — including both social networking and IAP via Facebook Payments — is written separately in C++ for iOS, C++ and Java for Android, and JavaScript for HTML5. All of these implementations are accessible from your game using write code for and test.

Likewise, the following code will display a full-screen video advertisement. Behind the scenes, this single JavaScript API call runs native C++, Java or JavaScript, depending on the platform.

Rapid Game Pro helps perform day-to-day development tasks faster. The Cocos2D X libraries and plugins are prebuilt, so when you run your game in the simulator or on the device it will launch almost instantaneously.

Developing your own game template with social networking, monetization and other plugins for multiple platforms — for even just one platform — can take months to get right. Rapid Game Pro lets you start with all the little things a pro-grade game needs already done.

For a game developer, choosing the right cross-platform game engine can be the single most important decision they make. I hope my insights help you to make that choice.

(b) Describe the 3D graphics pipelines in detail. Explain the various inputs to this pipeline and the operations performed on it by graphics pipeline. [10 Marks]

In 3D computer graphics, the graphics pipeline or rendering pipeline refers to the sequence of steps used to create a 2D raster representation of a 3D scene. Plainly speaking, once a 3D model has been created, for instance in a video game or any other 3D computer animation, the graphics pipeline is the process of turning that 3D model into what the computer displays.

In the early history of 3D computer graphics, fixed purpose hardware was used to speed up the steps of the pipeline through a fixed-function pipeline. Later, the hardware evolved, becoming more general purpose, allowing greater flexibility in graphics rendering as well as more generalized hardware, and allowing the same generalized hardware to perform not only different steps of the pipeline, like in fixed purpose hardware, but even in limited forms of general purpose computing.

As the hardware evolved, so did the graphics pipelines, the OpenGL, and DirectX pipelines, but the general concept of the pipeline remains the same. The 3D pipeline usually refers to the most common form of computer 3D rendering, 3D polygon rendering, distinct from ray tracing, and raycasting. In particular, 3D

polygon rendering is similar to raycasting. In raycasting, a ray originates at the point where the camera resides, if that ray hits a surface, then the color and lighting of the point on the surface where the ray hit is calculated.

In 3D polygon rendering the reverse happens, the area that is in view of the camera is calculated, and then rays are created from every part of every surface in view of the camera and traced back to the camera Computers began undergoing a significant change in recent years with the introduction of a separate video card and the rise of hardware accelerated graphics. This has led to the need for a programmable graphics pipeline which can be manipulated by shaders

Since the introduction of the programmable graphics pipeline most fixed-function pipeline implementations have become obsolete, such as OpenGL's immediate mode, or Direct3D's built in hardware Transform, clipping, and lightingThe Direct3D 11 programmable pipeline is designed for generating graphics for realtime gaming applications. This section describes the Direct3D 11 programmable pipeline. The following diagram shows the data flow from input to output through each of the programmable stages.

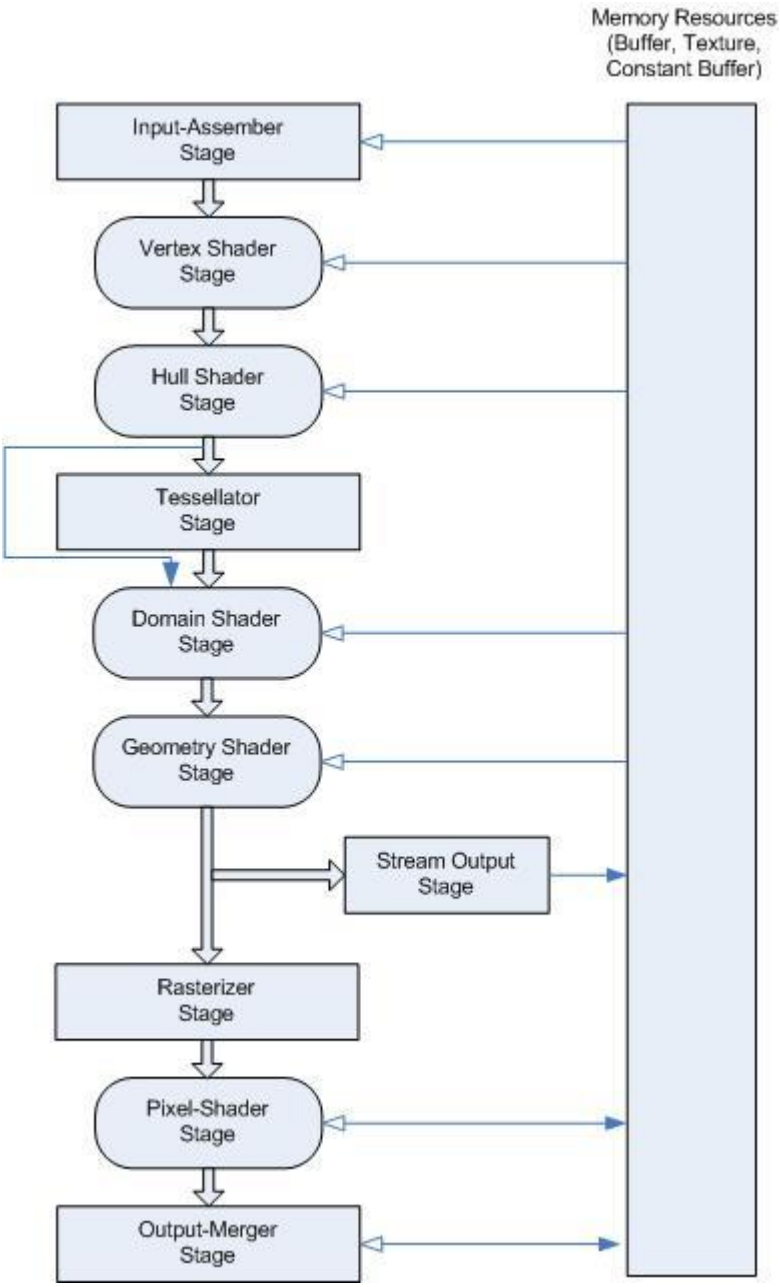


Figure : Graphics pipeline

The graphics pipeline for Microsoft Direct3D 11 supports the same stages as the Direct3D 10 graphics pipeline, with additional stages to support advanced features.

You can use the Direct3D 11API to configure all of the stages. Stages that feature common shader cores (the rounded rectangular blocks) are programmable by using the HLSL programming language. As you will see, this makes the pipeline extremely flexible and adaptable. The following list specifies the purpose of each of the stages.

- Input-Assembler Stage - The input-assembler stage supplies data (triangles, lines and points) to the pipeline.

- Vertex-Shader Stage - The vertex-shader stage processes vertices, typically performing operations such as transformations, skinning, and lighting. A vertex shader always takes a single input vertex and produces a single output vertex.

- Geometry-Shader Stage - The geometry-shader stage processes entire primitives. Its input is a full primitive (which is three vertices for a triangle, two vertices for a line, or a single vertex for a point). In addition, each primitive can also include the vertex data for any edge-adjacent primitives.

  This could include at most an additional three vertices for a triangle or an additional two vertices for a line. The geometry shader also supports limited geometry amplification and de-amplification. Given an input primitive, the geometry shader can discard the primitive, or emit one or more new primitives.

- Stream-Output Stage - The stream-output stage streams primitive data from the pipeline to memory on its way to the rasterizer. Data can be streamed out and/or passed into the rasterizer. Data streamed out to memory can be recirculated back into the pipeline as input data or read-back from the CPU.

- Rasterizer Stage - The rasterizer clips primitives, prepares primitives for the pixel shader, and determines how to invoke pixel shaders.

- Pixel-Shader Stage - The pixel-shader stage receives interpolated data for a primitive and generates per-pixel data such as color.

- Output-Merger Stage - The output-merger stage combines various types of output data (pixel shader values, depth and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.

- Hull-shader, tessellator, and domain-shader stages, which comprise the tessellation stages - The tessellation stages convert higher-order surfaces to triangles for rendering within the Direct3D 11 pipeline.

The Direct3D 11 programmable pipeline is also designed for providing high-speed computing tasks. A compute shader expands Direct3D 11 beyond graphics to support general purpose GPU computing.

7. Write short note on (Any four) :- [20 Marks]

a. Peek Message Method

b. Hard and soft architectures-

Architecture Design

When we begin to consider the actual architecture of the game (refer figure 2.1), we need also to consider how to construct it around the planning needs of the schedule. “Milestones and Deadlines”. Each milestone will specify the technical requirements to complete a particular tier. As we have been discussing, the architecture is specified in three main stages, each of which expands into a number of tiers.

The three stages are:

- Prototyping :

The prototyping stage allows us to have a dress rehearsal of the full architecture, allowing us to tackle any tricky points and difficulties that we might encounter. Of course, this doesn’t mean that we are going to be able to cover all of these difficulties, but we can tackle at least the more obvious ones, and, of course, we will be able to explore gameplay issues sooner than we would be able to otherwise.

- Hard-architecture design :

The hard-architecture design stage involves the laying of the game framework. However, the point of using a component- based design is to produce a set of generic components that can be used across projects. Hard-architecture components would need to be upgraded and augmented in order to keep up with emerging technology, but, with a sensible set of interfaces, the disruption caused by this continual upgrade (not replacement) process would be minimal. Only after a few projects have been undertaken will we see any benefits from reusing in-house components.

Once a few projects have been completed, we will also be able to use our own components.

- Soft-architecture design :

“The Software Factory,” we mentioned that the software factory architecture caused an apparent drop in productivity for the first project developed using it, by applying more structured development methods that reduce the amount of time spent on backtracking and unnecessary rework. This is pretty much unique for any project, and this is as it should be, for within the soft architecture is the unique spark that makes your game stand out from the rest. By using the soft-architecture system, we are taking advantage of all the groundwork that has already been done for us. The soft architecture defines the game-specific functionality and data required, such as in game graphics, music, and other data. In fact, the soft architecture is essentially data driven.

c. Chroma key-

Chroma key compositing, or chroma keying, is a special effects / post-production technique for compositing two images or video streams together based on color hues. The technique has been used heavily in many fields to remove a background from the subject of a photo or video – particularly the news-casting, motion picture and videogame industries. A color range in the top layer is made transparent, revealing another image behind. The chroma keying technique is commonly used in video production and post-production. This technique is also referred to as color keying, colour-separation overlay primarily by the BBC, or by various terms for specific color-related variants such as green screen, and blue screen – chroma keying can be done with backgrounds of any color that are uniform and distinct, but green and blue backgrounds are more commonly used because they differ most distinctly in hue from most human skin colors. No part of the subject being filmed or photographed may duplicate a color used in the background.

It is commonly used for weather forecast broadcasts, wherein a news presenter is usually seen standing in front of a large CGI map during live television newscasts, though in actuality it is a large blue or green background. When using a blue screen, different weather maps are added on the parts of the image where the color is blue. If the news presenter wears blue clothes, his or her clothes will also be replaced with the background video. A complementary system is used for green screens. Chroma keying is also used in the entertainment industry for special effects in movies and videogames. The advanced state of the technology and much commercially available computer software, such as Autodesk Smoke, Final Cut Pro, Pinnacle Studio, Adobe After Effects, and dozens of other computer programs, makes it possible and relatively easy for the average home computer user to create videos using the "chromakey" function with easily affordable green screen or blue screen kits.

d. Scene nodes-

A scene graph is a set of tree data structures where every item has zero or one parent, and each item is either a "leaf" with zero sub-items or a "branch" with zero or more sub-items.

Each item in the scene graph is called a Node. Branch nodes are of type Parent, whose concrete subclasses are Group, Region, and Control, or subclasses thereof.

Leaf nodes are classes such as Rectangle, Text, ImageView, MediaView, or other such leaf classes which cannot have children. Only a single node within each scene graph tree will have no parent, which is referred to as the "root" node.

There may be several trees in the scene graph. Some trees may be part of a Scene, in which case they are eligible to be displayed. Other trees might not be part of any Scene.

A node may occur at most once anywhere in the scene graph. Specifically, a node must appear no more than once in all of the following: as the root node of a Scene, the children ObservableList of a Parent, or as the clip of a Node.

The scene graph must not have cycles. A cycle would exist if a node is an ancestor of itself in the tree, considering the Group content ObservableList, Parent children ObservableList, and Node clip relationships mentioned above.

If a program adds a child node to a Parent (including Group, Region, etc) and that node is already a child of a different Parent or the root of a Scene, the node is automatically (and silently) removed from its former parent. If a program attempts to modify the scene graph in any other way that violates the above rules, an exception is thrown, the modification attempt is ignored and the scene graph is restored to its previous state.

It is possible to rearrange the structure of the scene graph, for example, to move a subtree from one location in the scene graph to another. In order to do this, one would normally remove the subtree from its old location before inserting it at the new location. However, the subtree will be automatically removed as described above if the application doesn't explicitly remove it.

Node objects may be constructed and modified on any thread as long they are not yet attached to a Scene. An application must attach nodes to a Scene, and modify nodes that are already attached to a Scene, on the JavaFX Application Thread.

e. Stack memory Vs Heap memory

The Stack- The stack is a "FILO" (first in, last out) data structure, which is managed and optimized by the CPU quite closely. Every time a function declares a new variable, it is "pushed" onto the stack. Then every time a function exits, all of the variables pushed onto the stack by that function, are freed (that is to say, they are deleted). Once a stack variable is freed, that region of memory becomes available for other stack variables.

The advantage of using the stack to store variables, is that memory is managed for you. You don't have to allocate memory by hand, or free it once you don't need it any more. What's more, because the CPU organizes stack memory so efficiently, reading from and writing to stack variables is very fast.

A key to understanding the stack is the notion that when a function exits, all of its variables are popped off of the stack (and hence lost forever). Thus stack variables are local in nature. This is related to a concept we

saw earlier known as variable scope, or local vs global variables. A common bug in C programming is attempting to access a variable that was created on the stack inside some function, from a place in your program outside of that function (i.e. after that function has exited).

Another feature of the stack to keep in mind, is that there is a limit (varies with OS) on the size of variables that can be store on the stack. This is not the case for variables allocated on the heap.

To summarize the stack:

- the stack grows and shrinks as functions push and pop local variables
- there is no need to manage the memory yourself, variables are allocated and freed automatically
- the stack has size limits
- stack variables only exist while the function that created them, is running

The Heap- The heap is a region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is a more free-floating region of memory (and is larger). To allocate memory on the heap, you must use `malloc ()` or `calloc()`, which are built-in C functions. Once you have allocated memory on the heap, you are responsible for using `free ()` to deallocate that memory once you don't need it any more. If you fail to do this, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside (and won't be available to other processes). As we will see in the debugging section, there is a tool called `valgrind` that can help you detect memory leaks.

Unlike the stack, the heap does not have size restrictions on variable size (apart from the obvious physical limitations of your computer). Heap memory is slightly slower to be read from and written to, because one has to use pointers to access memory on the heap. We will talk about pointers shortly.

Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

Stack verses Heap Pros and Cons

Stack

- very fast access
- don't have to explicitly de-allocate variables
- space is managed efficiently by CPU, memory will not become fragmented
- local variables only
- limit on stack size (OS-dependent)

- variables cannot be resized

Heap

- variables can be accessed globally
- no limit on memory size
- (relatively) slower access
- no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
- you must manage memory (you're in charge of allocating and freeing variables)
- variables can be resized using realloc().

f. Audio Formats-

An audio file format is a file format for storing digital audio data on a computer system. The bit layout of the audio data (excluding metadata) is called the audio coding format and can be uncompressed, or compressed to reduce the file size, often using lossy compression. The data can be a raw bitstream in an audio coding format, but it is usually embedded in a container format or an audio data format with defined storage layer.

There are three major groups of audio file formats:

Uncompressed audio formats, such as WAV, AIFF, AU or raw header-less PCM; Formats with lossless compression, such as FLAC, Monkey's Audio (filename extension .ape), WavPack (filename extension .wv), TTA, ATRAC Advanced Lossless, ALAC (filename extension .m4a), MPEG-4 SLS, MPEG-4 ALS, MPEG-4 DST, Windows Media Audio Lossless (WMA Lossless), and Shorten (SHN).

Formats with lossy compression, such as Opus, MP3, Vorbis, Musepack, AAC, ATRAC and Windows Media Audio Lossy (WMA lossy).

- Uncompressed audio format

One major uncompressed audio format, LPCM, is the same variety of PCM as used in Compact Disc Digital Audio and is the format most commonly accepted by low level audio APIs and D/A converter hardware. Although LPCM can be stored on a computer as a raw audio format, it is usually stored in a .wav file on Windows or in a .aiff file on Mac OS. The AIFF format is based on the Interchange File Format (IFF), and the WAV format is based on the similar Resource Interchange File Format (RIFF). WAV and AIFF are not inherently lossless; they're designed to store a wide variety of audio formats, lossless and lossy; they just add a small, metadata-containing header before the audio data to declare the format of the audio data, such as LPCM

with a particular sample rate, bit depth, number of channels. Since WAV and AIFF are widely supported and can store LPCM,

- Lossless compressed audio format

A lossless compressed format stores data in less space without losing any information. The original, uncompressed data can be recreated from the compressed version.

Uncompressed audio formats encode both sound and silence with the same number of bits per unit of time. Encoding an uncompressed minute of absolute silence produces a file of the same size as encoding an uncompressed minute of music. In a lossless compressed format, however, the music would occupy a smaller file than an uncompressed format and the silence would take up almost no space at all.

- Lossy compressed audio format

Lossy compression enables even greater reductions in file size by removing some of the audio information and simplifying the data. This of course results in a reduction in audio quality, but a variety of techniques are used, mainly by exploiting psychoacoustics, to remove the parts of the sound that have the least effect on perceived quality, and to minimize the amount of audible noise added during the process. The popular MP3 format is probably the best-known example, but the AAC format found on the iTunes Music Store is also common. Most formats offer a range of degrees of compression, generally measured in bit rate. The lower the rate, the smaller the file and the more significant the quality loss.

B.E 8th SEM IT JUNE -14 **MARKS-100**

1(a) Explain tokenization with any game example?

In general, a token is an object that represents something else, such as another object (either physical or virtual), or an abstract concept as, for example, a gift is sometimes referred to as a token of the. We can also consider these tokens to be arranged in a form of hierarchical structure.

The playing area, or game world, in itself is at the top of the hierarchy. From then on in, it is an essentially flat hierarchy.

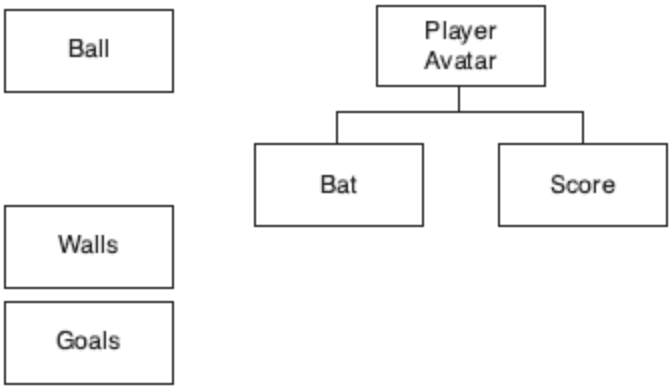


Figure : The Pong token hierarchy.

The game world token contains all the other tokens. Obviously every token has to operate within the game world in order to form a part of the game. The player avatar token is the representation of the player within the game world. It is effectively a channel for the user interface between the player and the game.

The player avatar for Pong is very simple; it is merely a bat and a score. These are how the player is represented in Pong. The other tokens—those manipulated by the computer—are the ball, the walls, and the goal zones. Now it's time for a little sleight of hand of the sort that is possible with only the written word. Reread the two paragraphs, and for every instance of the word "token," read it as "object."

So, if we were just talking about objects all along, why didn't we just use the word "object" to start with?

The main reason—and why we particularly like the use of the word token and why we will continue using it from here on in—is that these conceptual tokens may not have a one-to-one mapping with the programming language objects that are defined by the programmers. What we are trying to do is to break down the game design into conceptual objects that will eventually be translated into programming language objects. This tokenization process is an intermediary stage in the production of a decent architecture. In order to describe this without causing confusion, we need to use different terminology for each type of object.

The tokenization of a game design such as Pong is fairly trivial, and there's really only one way to do it. In spite of this, it makes an excellent example to try to demonstrate the thought processes behind tokenization. Not all games will be so trivial, and, for some more complex games, there may be

many ways, all of which are equally valid. So now we have a set of tokens. On their own, they are not very exciting as they do not interact with one another. We can now define an event—the collision event. Let's say that a collision event is generated when two tokens collide. The net result of this event is that each token receives a message telling it that a collision has occurred, and the type of object it has collided with.

The token interaction matrix is a very important construct. It is a chart of all the interactions that take place in the game. Note that for very large games we would not use a token-token matrix directly. Instead we would introduce an extra layer of abstraction by using token-property and/or property-property.

Okay, so let's look at the Pong token interaction matrix. The matrix is arranged in a triangular format, with each token listed along the side and the bottom. An unusual feature of Pong is that tokens do not come into contact with other tokens of the same type. This immediately means that the token-token interactions for bat-bat, ball-ball, wall-wall, goal-goal, and score-score can be discounted. Due to the nature of the game, the following interactions can also be discounted: bat-goal, wall-goal, score-bat, score-ball, and score-wall.

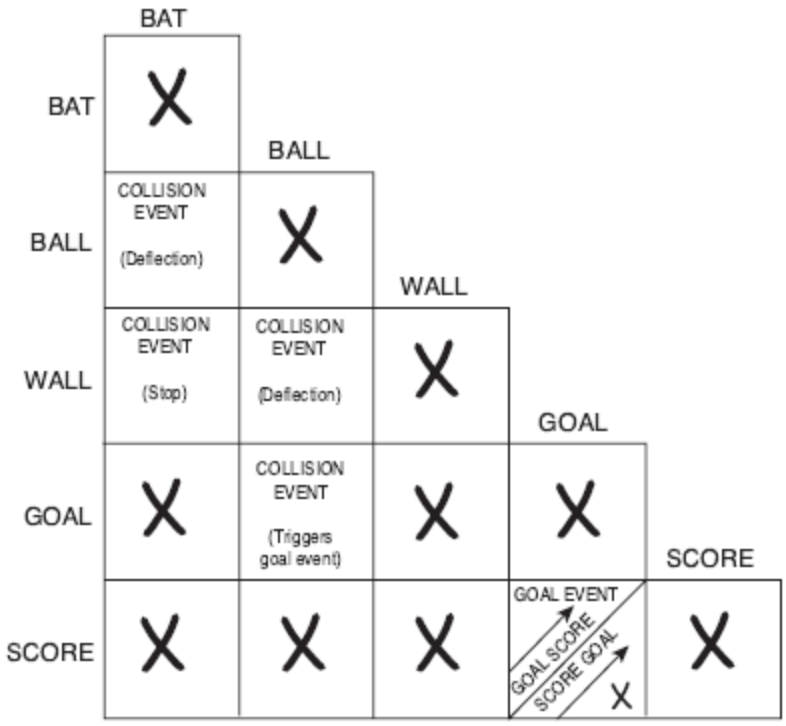


Figure : The Pong token interaction matrix

1(b) What is object factory explain in detail?

Object factories are one of those extremely useful but often times overlooked constructs in programming. To put it simply, object factories are like virtual functions but instead of dynamically choosing which function gets executed at runtime, object factories dynamically choose which class gets instantiated at runtime.

THE NEED - Here are just a few:

- Scripting language support – The game must decide which command class to create and execute based on text commands entered by the user.

- Serialization - One type of serialization would be communicating via TCP/IP. The receiving side must be able to dynamically create the proper message class depending on the type of message it has just received.

- Executing commands - Many games allow users to dynamically rebind keys to other commands. Pressing the 'A' key should be able to execute the command.

- To decreasing class dependencies – By not hard-coding the which class to instantiate we can greatly reduce class dependencies, as classes no longer need to know about other classes in order to create them. This can result in greatly decreased compile times.

The object factory is a class whose sole purpose is to allow the creation of families of objects. This means that the code creating the objects is not tied in specifically to the objects that it is creating. Usually, all the objects created by the factory derive from the same abstract base class, and are returned to the requesting client as a reference to this class. You could also have multiple methods, Make XXXX(), each returning a different base class, but personally we prefer to have one factory per object family.

For example, in *Balls!*, all of the tokens are created by an object factory. Among other things, this makes the loading of levels very simple. Each token type has a unique ID, and loading a level is a

simple matter of passing that ID to the object factory, and making use of the pointer returned. One of the other advantages of this is low-cost garbage collection and memory tracking.

The object factory can keep track of all the objects that it allocates, inserting them into a list. When these objects are deleted, they remove themselves from this list. If any of these objects have not been freed at the end of the level, then they can either be cleaned up automatically or reported as errors. This helped us find quite a few insidious little bugs in the code.

Figure shows the class relationships. This describes how the objects are related to each other programmatically. In Figure “game object factory” receives requests from clients to create a specific “game object.” This factory class is responsible for creating all objects representing tokens within a level, and keeps track of them by using a related class, the “memory tracker” class. The factory class produces instantiations of the requested object, which are returned to the client as a pointer to the base class of all in-game token classes, the game object class.

The game object class contains a memory tracker class as an instantiated member variable. When a game object class is instantiated, the memory tracker member is also instantiated as part of it. When instantiated, the constructor of this class stores a reference to its owner (the enclosing game object), and inserts itself into the list within the factory class.

When the enclosing game object is deleted, the destructor of the member memory tracker is invoked, which removes the reference to itself from the list within the factory class. When the factory object is itself destroyed, it checks through the list to see if any remaining game objects have not been deleted. These can be either freed automatically, or (preferably) reported as errors, to be tracked down later.

This sort of dynamic object creation is very flexible: With judicious use of the object factory, the structure of whole applications can be determined at runtime. As long as all the objects returned by a specific factory conform to the same interface and behavioral contracts, then it is possible to (for example) configure levels dynamically with fairly simple code, allow customization of the user interface, release game expansions that don’t need to modify the original game code, or accomplish any number of tasks that would be tricky by other means.

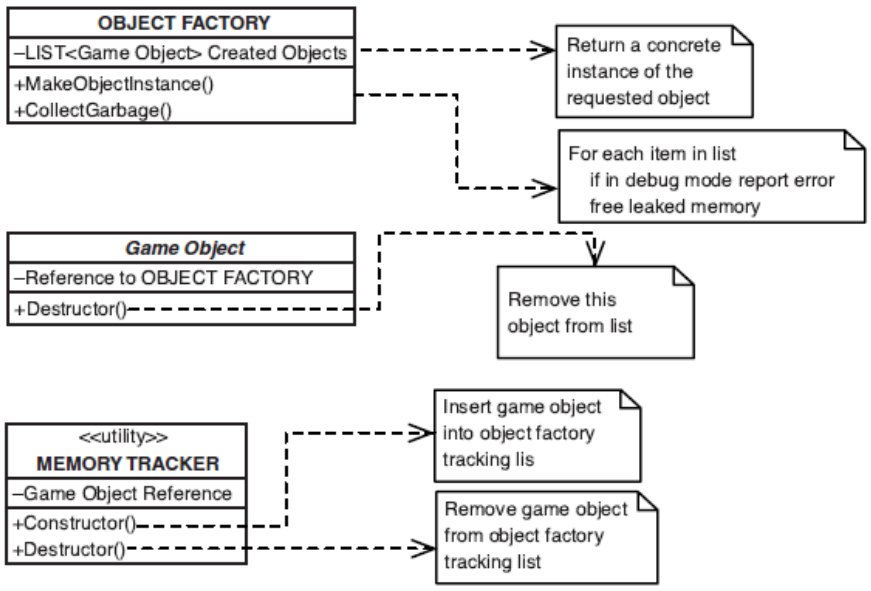


Figure : class relationships for the game object factory.

2(a) Explain the use of DIRECTX in game development?

Microsoft DirectX is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming and video, on Microsoft platforms. Originally, the names of these APIs all began with Direct, such as Direct3D, DirectDraw, DirectMusic, DirectPlay, DirectSound, and so forth. The name Direct*X* was coined as shorthand term for all of these APIs and soon became the name of the collection. When Microsoft later set out to develop a gaming console, the X was used as the basis of the name Xboxto indicate that the console was based on DirectX technology. The X initial has been carried forward in the naming of APIs designed for the Xbox such as XInput and the Cross-platform Audio Creation Tool .

Direct3D (the 3D graphics API within DirectX) is widely used in the development of video games for Microsoft Windows, Sega Dreamcast, Microsoft Xbox, Microsoft Xbox 360, and Microsoft Xbox One. Direct3D is also used by other software applications for visualization and graphics tasks such as CAD/CAM engineering. As Direct3D is the most widely publicized component of DirectX, it is common to see the names "DirectX" and "Direct3D" used interchangeably.

The DirectX software development kit (SDK) consists of runtime libraries in redistributable binary form, along with accompanying documentation and headers for use in coding. Originally, the runtimes were only installed by games or explicitly by the user. Windows 95 did not launch with DirectX, but DirectX was included with Windows 95 OEM Service Release 2.[2] Windows

98 and Windows NT 4.0 both shipped with DirectX, as has every version of Windows released since. The SDK is available as a free download. While the runtimes are proprietary, closed-source software, source code is provided for most of the SDK samples. Starting with the release of Windows 8 Developer Preview, DirectX SDK has been integrated into Windows SDK.

DirectX gives the highest fidelity and richest experiences in 3D gaming. DirectX supports a wide range of graphics feature levels, from DirectX 9.1 to all the latest hardware features exposed in DirectX 11 and 12. DirectX allows you to tailor your game to every PC, from power-efficient ARM-based portable tablets, to over-clocked multi-GPU gamer rigs.

2(b) What is hard and soft architectures. Explain in brief?

Architecture Design

When we begin to consider the actual architecture of the game (refer figure 2.1), we need also to consider how to construct it around the planning needs of the schedule. “Milestones and Deadlines”. Each milestone will specify the technical requirements to complete a particular tier. As we have been discussing, the architecture is specified in three main stages, each of which expands into a number of tiers.

The three stages are:

- Prototyping :

The prototyping stage allows us to have a dress rehearsal of the full architecture, allowing us to tackle any tricky points and difficulties that we might encounter. Of course, this doesn’t mean that we are going to be able to cover all of these difficulties, but we can tackle at least the more obvious ones, and, of course, we will be able to explore gameplay issues sooner than we would be able to otherwise.

- Hard-architecture design :

The hard-architecture design stage involves the laying of the game framework. However, the point of using a component- based design is to produce a set of generic components that can be used across projects. Hard-architecture components would need to be upgraded and augmented in order to keep up with emerging technology, but, with a sensible set of interfaces, the disruption caused by this continual upgrade (not replacement) process would be minimal. Only after a few projects have been undertaken will we see any benefits from reusing in-house components.

Once a few projects have been completed, we will also be able to use our own components.

- Soft-architecture design :

"The Software Factory," we mentioned that the software factory architecture caused an apparent drop in productivity for the first project developed using it, by applying more structured development methods that reduce the amount of time spent on backtracking and unnecessary rework. This is pretty much unique for any project, and this is as it should be, for within the soft architecture is the unique spark that makes your game stand out from the rest. By using the soft-architecture system, we are taking advantage of all the groundwork that has already been done for us. The soft architecture defines the game-specific functionality and data required, such as in game graphics, music, and other data. In fact, the soft architecture is essentially data driven.

3(b) Why is coding phase important? Explain coding priorities that need to be established as part of technical design?

4(a) What are the three stages of running a game? Explain in detail?

4(b) Explain in detail cleanup process to be followed during and after game exit?

5 (a) Explain source control system. [10 Marks]

A component of software configuration management, version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number," "revision level," or simply "revision." For example, an initial set of files is "revision 1." When the first change is made, the resulting set is "revision 2," and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated, when the era of computing began. The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may change the same files.

Version control systems (VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, e.g., Google Docs and Sheets[2] and in various content management systems, e.g., Wikipedia's Page history. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming.

Software tools for revision control are essential for the organization of multi-developer projects.

In computer software engineering, revision control is any kind of practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code.

As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops).

Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently (for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used on many large software projects. While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of self-discipline on the part of developers, and often leads to mistakes. Consequently, systems to automate some or all of the revision control process have been developed.

Moreover, in software development, legal and business practice and other environments, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary interests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even indispensable in such situations.

Structure revision control manages changes to a set of data over time. These changes can be structured in various ways.

Often the data is thought of as a collection of many individual items, such as files or documents, and changes to individual files are tracked. This accords with intuitions about separate files, but causes problems when identity changes, such as during renaming, splitting, or merging of files. Accordingly, some systems, such as git, instead consider changes to the data as a whole, which is less intuitive for simple changes, but simplifies more complex changes.

When data that is under revision control is modified, after being retrieved by *checking out*, this is not in general immediately reflected in the revision control system (in the *repository*), but must instead be *checked in* or *committed.* A copy outside revision control is known as a "working copy". As a simple example, when editing a computer file, the data stored in memory by the editing program is the working copy, which is committed by saving.

Concretely, one may print out a document, edit it by hand, and only later manually input the changes into a computer and save it. For source code control, the working copy is instead a copy of all files in a particular revision, generally stored locally on the developer's computer in this case saving the file only changes the working copy, and checking into the repository is a separate step.

If multiple people are working on a single data set or document, they are implicitly creating branches of the data (in their working copies), and thus issues of merging arise, as discussed below. For simple collaborative document editing, this can be prevented by using file locking or simply avoiding working on the same document that someone else is working on.

Revision control systems are often centralized, with a single authoritative data store, the *repository*, and check-outs and check-ins done with reference to this central repository. Alternatively, in distributed revision control, no single repository is authoritative, and data can be checked out and checked into any repository. When checking into a different repository, this is interpreted as a merge or patch.

5. (b) Explain various platforms on which game can be deployed on. What are the advantages and disadvantages of each of these platform? [10 Marks]

Finding the right game engine can be the key to successfully building and deploying a game that becomes both popular and lucrative. But there are so many game engines out there vying for your attention. Clearly, some guidelines on the subject would be useful.

Ten years ago, it was okay to release your game on one platform at a time. Today, it's more typical for a game to be released rapidly on multiple platforms. To that end, a cross-platform game engine offers some real advantages, and the options there are quite diverse and plentiful. Having recently released my own game template based on Cocos2D JS, I thought it would be interesting to compare some of the major game engines and see how they stack up against each other.

To prepare for this post, I wrote a complete Breakout clone in four of today's top cross-platform game engines: Unity, Corona, Cocos2D JS and Appcelerator Titanium, and also using my game template, RapidGame Pro. The code can be found at the end of each section, so you can see for yourself. My observations on how they all compare should help you make a choice that may save you and your team weeks or months.

Unity : Unity is, in short, a closed-source, cross-platform game development application. You create your game by manipulating objects in 3D and attaching various components to them. Even 2D games must be manipulated in 3D. Scripts are written in C# (recommended), Boo or Unityscript (mistakenly called JavaScript) and attached to 3D objects as components.

Launching Unity for the first time, you may feel like the pilot of a 747 jet plane. There is much to learn before even the first switch can be flipped. First of all, there's camera and lights. When trying to add a simple cube to the scene, it can get lost behind the camera or perhaps be invisible because there's no light. In short, there is a learning curve.

Prepare to spend approximately 8-12 hours getting familiar enough to develop your own game.Unity was first released in 2005 and the interface hasn't changed much since. To be frank, it feels like many of the repetitive tasks in day-to-day Unity game development are busy work. Adding audio sources, updating prefabs and importing assets are all examples of tasks that shouldn't have to be done, or shouldn't take so long. It would be nice if Unity had a modern makeover.

That said, once you've created a game with Unity, deployment is a cinch. With a couple of clicks, you can export your game to mobile, desktop and web currently requires the Unity player app to be installed. If you have the right license, you can even deploy to gaming consoles like Xbox.

Corona : Corona is a closed-source 2D game simulator and cloud-build application. Game code is written in Lua scripts and played back in the Corona simulator. Like Mystique from X-Men, the simulator can take on many skins, resolutions and ratios. When you're ready to deploy, it builds your game in the cloud and delivers you an iOS or Android game client.

Ah, the sweet joy of developing games with Corona. Everything about the language is easy. Adding a physics body, for example, takes only one line of code. After a mere 2-4 hours of getting familiar with the platform you'll be ready to develop games. And once you start it's difficult to stop. The simulator is responsive, quick and polite about using your computer's resources. With the simulator and your choice of code editor open side by side, you can save the Lua file and the simulator instantaneously reloads the game. It's simply delightful to develop a game with such rapidity.

One shortcoming of Corona is its limited deployment options. Only mobile platforms like iOS, Android, Kindle and Nook are supported. Windows Phone is coming soon. Cloud-Imagine a day full of testing your game on the device, tweaking one little thing and waiting a few minutes to be able to see if it worked.

Like Unity, Corona is closed-source and proprietary. There's no way to make a modification or fix a bug in the engine, and you cannot learn from its code.

Cocos2D JS : Cocos2D JS is a cross-platform, open-source, free game development SDK. It is the newest — and perhaps sexiest — member of the Cocos2D family. Essentially it's a combination of two popular open-source projects: Cocos2D X for mobile / desktop and Cocos2D HTML5 for web. While it is currently 2D / 2.5D, there are plans to add 3D support.

You write game code entirely in JavaScript. On native platforms like mobile and desktop, your game's JavaScript is bound to native C++ objects, granting you maximum speed without having to write any native code. Web platforms run pure JavaScript and render using Canvas or WebGL, so no player applications need to be installed.

The easiest way to get started with Cocos2D JS game development is using the HTML5 platform. Open up a browser window and your favorite text or code editor, save your JavaScript, refresh the browser and voila. It's a rapid way to develop. When you're ready to test and deploy to native platforms, you'll need Xcode, Visual Studio and/or Eclipse.

Cocos2D JS games can currently be deployed to iOS, Android, Blackberry, Windows Phone, Mac, Windows, Linux and HTML. With such wide deployment options, it's easy to see why many game developers are choosing Cocos2D.

Appcelerator Titanium : Titanium is a cross-platform, open-source app development kit and Eclipse-based IDE. Apps are written in JavaScript and run natively, not just in a WebView. With Titanium Studio it's possible to develop, test and deploy to mobile and web platforms.

For 2D game development, there's the Platino Game Engine, an open-source but not free — SDK that can be added to your Titanium stack. Getting acquainted with Titanium is not as easy as it could be. The documentation has holes. For example, the crucial .center sprite property is left undocumented. Moreover, the physics engine is cumbersome and archaic. You have to synchronize all physics bodies and sprites manually using a very non-JavaScript, C-like API.

On the bright side, one nice thing about Titanium development is that the SDK is prebuilt. You can run your game on a simulator or device with very short build times.

RapidGame Pro : RapidGame Pro, an open-core game (dual MIT licensed) template based on Cocos2D JS, to make game development using open source more rapid. It achieves this in a few ways:

By providing a project creator tool and game templates that make starting a game with scenes, sprites, sound, physics, a server, monetization, social, etc. a breeze.By prebuilding native libraries.

By providing and incorporating plugins for IAP, displaying ads, social networking,analytics, asynchronous multiplayer and virtual economies that work on all platforms.By including example code to a complete game based on multiple currencies.

Some of RapidGame Pro's plugins had to be developed from scratch for multiple platforms. For example, the Facebook plugin — including both social networking and IAP via Facebook Payments — is written separately in C++ for iOS, C++ and Java for Android, and JavaScript for HTML5. All of these implementations are accessible from your game using write code for and test.

Likewise, the following code will display a full-screen video advertisement. Behind the scenes, this single JavaScript API call runs native C++, Java or JavaScript, depending on the platform.

RapidGame Pro helps perform day-to-day development tasks faster. The Cocos2D X libraries and plugins are prebuilt, so when you run your game in the simulator or on the device it will launch almost instantaneously.

Developing your own game template with social networking, monetization and other plugins for multiple platforms — for even just one platform — can take months to get right.RapidGame Pro lets you start with all the little things a pro-grade game needs already done.

For a game developer, choosing the right cross-platform game engine can be the single most important decision they make. I hope my insights help you to make that choice.

6. (a) explain game play research in detail?

The gameplay could have an impact on the technology needed for the game. For example,the game's user interface may require investigation of certain types of controller. Other aspects of the gameplay may require active research. For example, in the case of strategy games, plenty of information is available if you're prepared to do a little digging: Game theory from the War Studies Group is readily available on the Web, as are analyses commissioned by the U.S. Navy and the Pentagon. Looking at history also indicates where different factors have contributed to the payoff matrices of a real situation, such as why the Aztecs became so powerful, and how in spite of this, they were beaten by Cortés.

For researching the gameplay of a puzzle game like Tetris or Balls!, you could look at psychological work on types of reasoning. A satisfying puzzle game like Tetris or Puzzle Bobble will include spatial and temporal reasoning (the "story" part) as well as logical reasoning using the manipulation of abstract concepts mapped to concrete entities (the "planning" part) and the pleasurable payoff of watching where all that leads, and learning something more about the way the game rules operate (the "play" or "learning curve" part.)

Obviously, you can find more genres of games, but the point we are making here is that you don't necessarily have to look in the standard or clichéd places for gameplay ideas, such as novels, films, and other games. There is a whole world of information out there, and a lot of it can be applied to gameplay, even if it does not seem immediately obvious. After all, the idea for Tetris was drawn from the field of mathematics.

The last kind of research involves researching the technology that is required to actually implement the game. Are you going to be using any new techniques, or breaking into uncharted waters?

This sort of research is what id Software spends most of its time doing. id Software has lots of money. This is no coincidence: without lots of cash backing you up, research has to be more focused and specific. You can't just go wandering through your ideas randomly, idly daubing code "pan-it" on the "canvas" of your compiler without someone being willing to pay the bills.

The fact is that this sort of research takes a lot of time and a lot of money. This is where the real research is, and where the bulk of the budget of time and money will be invested. In an ideal world, your company would have enough money to allow unrestricted research into new technology.

Unfortunately, unless you are id Software, then this is very unlikely. There will always be commercial pressures breathing down your necks, and the company management will be expecting results. In previous chapters, we have pointed out the dangers of depending on concrete results from research. I'll not repeat those warnings here.

Research is an unpredictable activity, and research into new technologies is particularly difficult. For every Quake engine, there are probably hundreds of failed attempts. Worse still, a fair proportion of these failed attempts will have been for game projects that had to be canned due to that failure: For the original release of Quake, the focus was on the technology and not so much on the game.

In contrast, Quake II was much less of a technological advance over Quake than Quake was over Doom, A few refinements were made to the engine, but most of the effort appeared to have gone into refining the gameplay and the storyline. Quake III Arena seems to have reverted to the original Quake model: concentrating on the gameplay emerging from the technology (which has been improved by the addition of quadratic curve rendering).

This is an interesting approach that appears to defy conventional game theory. Even the single-player game is a multiplayer game, with all the other players controlled by the computer. I'd go as far to say that there is no real gameplay in a multiplayer game: It's more a simulation set in a fantasy

environment. It's too close to reality (albeit a fictional reality) for it to be considered a game. It's an accurate simulation of future combat.

You may be getting the impression that we are against technology in some way. That's not true. We are against the gratuitous use of technology in the same way as we are against gratuitous violence. It's unnecessary and is, in some cases, quite disturbing.

The whole industry (with few exceptions) appears to be putting technology before gameplay, and this is a dismal and worrying prospect. We've lost count of the number of games we have seen that have lost gameplay value because the developers wanted to showcase their latest technologies. Case Studies 18.3 and 18.4 gives details of a much-welcome exception to this pattern.

Research should be treated with as much seriousness as you would find in a laboratory. Everything should be documented. Every thought, every procedure, and every result— even the wrong ones—needs to be recorded. This is serious stuff. Research is the lifeblood of your company. You need to research in order to keep up with the fast changing pace of technology. If not, you risk being left behind in the rush.

6 (b) discuss the content of game design document?

A game design document is a highly descriptive living design document of the design for a video game. A GDD is created and edited by the development team and it is primarily used in the video game industry to organize efforts within a development team. The document is created by the development team as result of collaboration between their designers, artists and programmers as a guiding vision which is used throughout the game development process. When a game is commissioned by a game publisher to the development team, the document must be created by the development team and it is often attached to the agreement between publisher and developer; the developer has to adhere to the GDD during game development process.

You must consider two facts:

➢ The gameplay spec needs to be available to your programmers.

➢ The programmers will not read the gameplay spec.

They really won't. You can tell them it's vitally important. You can plead with them. You can even offer royalties. But the programmers will not read that spec. You may have heard the saying that programmers can't see the wood for the trees, while designers can't see the trees for the wood. It's true, except that programmers sometimes can't even see the trees for the leaves on the trees. This is what makes them good at programming, the ability to break things down to quantum levels of each tiny individual step in a process. It's also why they aren't going to take a long document like the gameplay spec and read and absorb it to the point where they've built a model of the whole finished product in their heads.

But the programming team does need the design documentation readily available for several reasons. First, carrying out tasks without knowing the goal is demoralizing and counterproductive. Second, when some aspect of a task is open to question, it's more efficient to be able to refer to the spec than it is to hold a meeting to answer it. Third, although it is financially expedient to have only a single person or small group write the spec, it's beneficial to involve everybody's best ideas in evolving it to perfection. Last, a shared vision of the project contributes to group cohesion and morale. It's a dilemma. They need it, but they won't read it. Obviously, the design must be made accessible to the programmers in some other way. You could permanently assign a member of the design group to sit guru-like atop a mountain of 3D Studio MAX boxes and answer questions whenever they arose. Because it's unlikely that this would prove to be cost effective, another solution that is almost as good is to have the design documentation on an intranet site.

A game design document may be made of text, images, diagrams, concept art, Concept art is a form of illustration used to convey an idea for use in films, video games, animation, comic books or other media before it is put into the final product.[1] Concept art is also referred to as visual development and/or concept design. This term can also be applied to retail, set, fashion, architectural and industrial design.

Concept art is developed in several iterations. Artists try several designs to achieve the desired result for the work, or sometimes searching for an interesting result. Designs are filtered and refined in stages to narrow down the options. Concept art is not only used to develop the work, but also to show the project's progress to directors, clients and investors. Once the development of the work is complete, advertising materials often resemble concept art, although these are typically made specifically for this purposed, based on final work or any applicable media to better illustrate design decisions.

Some design documents may include functional prototypes Software prototyping is the activity of creating prototypes of software applications, i.e., incomplete versions of the software program being developed. It is an activity that can occur in software development and is comparable to prototyping as known from other fields, such as mechanical engineering or manufacturing.

A prototype typically simulates only a few aspects of, and may be completely different from, the final product.

Prototyping has several benefits: The software designer and implementer can get valuable feedback from the users early in the project. The client and the contractor can compare if the software made matches the software specification, according to which the software program is built. It also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met. The degree of completeness and the techniques used in the prototyping have been in development and debate since its proposal in the early 1970s or a chosen game engine

A game engine is a software framework designed for the creation and development of video games. Developers use them to create games for consoles, mobile devices and personal computers. The core functionality typically provided by a game engine includes a rendering engine (“renderer”) for 2D or 3D graphics, a physics engine or collision detection (and collision response), sound, scripting, animation, artificial intelligence, networking, streaming, memory management, threading, localization support, and a scene graph. The process of game development is often economized, in large part, by reusing/adapting the same game engine to create different games, or to make it easier to "port" games to multiple platforms for some sections of the game.

Although considered a requirement by many companies, a GDD has no set industry standard form. For example, developers may choose to keep the document as a word processed document, Concept art is a form of illustration used to convey an idea for use in films, video games, animation, comic books or other media before it is put into the final product. Concept art is also referred to as visual development and/or concept design. This term can also be applied to retail, set, fashion, architectural and industrial design.

Concept art is developed in several iterations. Artists try several designs to achieve the desired result for the work, or sometimes searching for an interesting result. Designs are filtered and refined in

stages to narrow down the options. Concept art is not only used to develop the work, but also to show the project's progress to directors, clients and investors. Once the development of the work is complete, advertising materials often resemble concept art, although these are typically made specifically for this purposed, based on final work or as an on-line collaboration tool. Collaborative software or groupware is an application software designed to help people involved in a common task to achieve their goals. One of the earliest definitions of collaborative software is 'intentional group processes plus software to support them.

Collaborative software is a broad concept that overlaps considerably with computer-supported cooperative work (CSCW) groupware is part of CSCW. The authors claim that CSCW, and thereby groupware, addresses "how collaborative activities and their coordination can be supported by means of computer systems." Software products such as email, calendaring, text chat, wiki, and bookmarking belong to this category whenever used for group work, whereas the more general term social software applies to systems used outside the workplace, for example, online dating services and social networking sites like Twitter and Facebook.

6 a) Explain the various platforms on which game can be deployed on? What are the advantages and disadvantages of each of these platforms?

Unity- Unity is a cross-platform game development engine that allows developers to create their games through the manipulation of objects in 3D. The closed-source system has various components attached to those 3D objects. C#, Unity-Script or Boo are used to write the scripts. Released in 2005, Unity is an interface that hasn’t changed much. In some ways, it features tasks that are too repetitive and that may hinder with the overall game development-process.

Once a game is created with Unity, the development that follows afterwards is a simple task. A few clicks are required to export your creation to web, desktop or mobile. Unity has a functional free version available, although if you’re an avid game developer, the paid version is a much better option because it allows setup to major platforms. Unity is leading multi platform game engine with many developers using it.

Corona is a cloud-build app and 2D game simulator. The game code uses Lua scripts for the writing, and just like X-Men and Mystique, the simulator can afford various skins, ratios and resolutions. When the developer is all set for installation, the game is created in the cloud. Featuring a

polite, responsive and quick simulator, Corona's sole shortcoming is the limited deployment options. The platform can only accept Android, iOS, Kindle, and Nook. According to the creators, soon enough they'll also make the platform available for Windows Phone. There's free versions available (Starter), but also various paid versions with yearly subscriptions.

Cocos2D JS is a free-development SDK, open-source cross-platform. Experts agree that it's one of the coolest on the market. Basically, it is a combo of 2 incredibly popular open-source projects: Cocos2D HTML5 (web) and Cocos2D X (mobile). JavaScript is used to write the code of the game. On desktop and mobile, the JavaScript of your future game is bound to C++ objects (native), thus providing maximum speed with no need to write native code. Using you are advised to use HTML5 for game development; as soon as you're all set to test and launch to native platforms, Eclipse, Xcode or Visual Studio will be needed. One of the greatest benefits of Cocos2D JS is its open-source feature. A lot of things can be learned just by interpreting the code.

Appcelerator Titanium is an open-source, cross-platform, app development kit. This is one of the most popular mobile game development platforms in recent past. JavaScript is used to write the apps. By using this game development engine, you'll be able to create, deploy and test your invention on web platforms and mobile. It's great that Titanium's SDK is prebuilt; thus, your game can be run on a device or simulator with limited build intervals.

WGame mainly functions on top of marmalade and it can be deployed at mobiles and desktops with Lua. It is an open-source, free and proprietary software which runs on different operating systems such as Windows, Linux etc. It is primarily used for developing video games.

SIO2 can be broadly defined as OpenGLES based platform game development for 2D and 3D game which operates on various operating systems such as Windows, MacOS, Android and iOs. This game development tool also enables you to port the game at Mac store as well as Windows.

The loom engine uses the AS3- like ECMAScript but does not build native code. But it uses Cocos2D workflow which allows you to take your time while building the game. It just requires 1 commend for making a new project and second command to run it effectively. As it is open-source, so the developer can get it for free.

SDL or Simple DirectMedia Layer is one of the most popular cross-development libraries which have been designed especially as it offers low level access to joystick, mouse, keyboard, and audio. It

offers access to graphics hardware via Direct3D and OpenGl. It supports various OS such as Android, iOS, Linux, MacOS X and Windows.

SFML offers a simple interface for various components of PC for making the development process of multimedia and games applications. There are five modules which are networking, audio, graphics, window and system. This is a multiplatform program which compiles and run on various operating systems such as Windows, Mac OS X and Linux.

GameMaker game development tools allow both entry level novices and experienced game development professionals for the creation of cross-platform games with less cost and record time. The GameMaker also allows developers for the creation of fully functional prototypes in some hour's time. You can also create complete games in some weeks.

Benefits of Developing for Multiple Platforms:

As is obvious, the more platforms you cover, the more people you'll be able to reach. With Apple's iOS and Google's Android competing for top positions worldwide, the number of smartphone users for these systems is increasing day by day. Developing an application that runs on both iPhone and Android gives you the added advantage of tapping into greater market potential.

EASY MARKETING- When you have a larger fan base, marketing becomes easier in the sense that you don't have to create niche messages to cater to a specific set of people. You have the liberty of marketing the application on various media and through generalised messages for the masses.

ONE INSTEAD OF MANY- It is easier to maintain and deploy changes when you're developing one application that runs across all platforms. Updates would immediately get synced across all devices and platforms. Further, with tools like Appcelerator and PhoneGap, it becomes easy to handle one team of developers working on a single multi-platform app than several teams working on different platforms.

UNIFORM LOOK AND FEEL -The overall design and feel of the app can be maintained across various platforms if there's a single code running on all. When you're designing different apps, it can be hard to sync two different developers or teams of different levels of expertise.

USE OF KNOWN TECHNOLOGIES- When you're using tools like Appcelerator, you can easily code in HTML5 and convert for different mobile platforms. This means you're using resources you already know about and converting them for deployment across different platforms. This leads us to our next point…

HTML5 VS. MOBILE DEVELOPERS- It is harder to find mobile developers but relatively easy to find good HTML, CSS and JavaScript coders. If you're using HTML5, this means hiring for development can be easier if you're designing cross-platform apps.

REDUCED DEVELOPMENT COSTS- All this means you have reduced development costs when making apps for multiple platforms. But before you make a decision, read on for the disadvantages…

Disadvantages of Developing for Multiple Platforms:

USER INTERACTION- iPhone and Android alone have significantly diverse screen layouts. Designing one app that fits on both these and more platforms can be quite a task.

PLATFORM INTEGRATION- It's not just the UI that is different. When it comes to integrating the app with the local settings, preferences and notifications apps, you can be faced with serious trouble trying to juggle multiple platforms. Even storage options are diversified so you may be looking at cloud options and integration of third party cloud services with your app.

TRYING TO PLEASE EVERYONE- According to Christina Warren, you could be faced with the same dilemma when developing an app for multiple platforms. She says, "a good cross-platform application looks at home on whatever platform it is used on. A bad cross-platform tries to look identical everywhere." So it's one thing being the same, and another being similar. You can't be the same on every platform, but have to adapt to each platform's unique styles – a functionality you lose if you're creating one app for all.

LOSS OF FLEXIBILITY- Each platform provides its own flexibilities – that's why they're there on the market. When you're designing a cross-platform app, you're forced to look at the commonalities. This puts you at a disadvantage of losing the flexibility that each platform provides.

5 b) What is source control? Explain in brief the different functionalities provided by Source Control System?

A component of software configuration management, version control, also known as revision control or source control,[1]:2 is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number," "revision level," or simply "revision." For example, an initial set of files is "revision 1." When the first change is made, the resulting set is "revision 2," and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated, when the era of computing began. The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may change the same files.

Version control systems (VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, e.g., Google Docs and Sheets[2] and in various content management systems, e.g., Wikipedia's Page history. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming.

Software tools for revision control are essential for the organization of multi-developer projects.

In computer software engineering, revision control is any kind of practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code.

As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops).

Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently (for instance, where one

version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used on many large software projects. While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of self-discipline on the part of developers, and often leads to mistakes. Consequently, systems to automate some or all of the revision control process have been developed.

Moreover, in software development, legal and business practice and other environments, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary interests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even indispensable in such situations.

Structure revision control manages changes to a set of data over time. These changes can be structured in various ways.

Often the data is thought of as a collection of many individual items, such as files or documents, and changes to individual files are tracked. This accords with intuitions about separate files, but causes problems when identity changes, such as during renaming, splitting, or merging of files. Accordingly, some systems, such as git, instead consider changes to the data as a whole, which is less intuitive for simple changes, but simplifies more complex changes.

When data that is under revision control is modified, after being retrieved by checking out, this is not in general immediately reflected in the revision control system (in the repository), but must instead be checked in or committed. A copy outside revision control is known as a "working copy". As a simple example, when editing a computer file, the data stored in memory by the editing program is the working copy, which is committed by saving.

Concretely, one may print out a document, edit it by hand, and only later manually input the changes into a computer and save it. For source code control, the working copy is instead a copy of all files in a particular revision, generally stored locally on the developer's computer in this case saving the file only changes the working copy, and checking into the repository is a separate step.

If multiple people are working on a single data set or document, they are implicitly creating branches of the data (in their working copies), and thus issues of merging arise, as discussed below. For simple collaborative document editing, this can be prevented by using file locking or simply avoiding working on the same document that someone else is working on.

Revision control systems are often centralized, with a single authoritative data store, the repository, and check-outs and check-ins done with reference to this central repository. Alternatively, in distributed revision control, no single repository is authoritative, and data can be checked out and checked into any repository. When checking into a different repository, this is interpreted as a merge or patch.

The benefits of source control

- Easier backups of one, central location
- Easy development of new features
- Historical overview of changes
- Access control for revisions.

7 (a) short note –

(a) Audio formats-

An audio file format is a file format for storing digital audio data on a computer system. The bit layout of the audio data (excluding metadata) is called the audio coding format and can be uncompressed, or compressed to reduce the file size, often using lossy compression. The data can be a raw bit-stream in an audio coding format, but it is usually embedded in a container format or an audio data format with defined storage layer.

There are three major groups of audio file formats:

Uncompressed audio formats, such as WAV, AIFF, AU or raw header-less PCM; Formats with lossless compression, such as FLAC, Monkey's Audio (filename extension .ape), WavPack (filename extension .wv), TTA, ATRAC Advanced Lossless, ALAC (filename extension .m4a), MPEG-4 SLS, MPEG-4 ALS, MPEG-4 DST, Windows Media Audio Lossless (WMA Lossless), and Shorten (SHN).

Formats with lossy compression, such as Opus, MP3, Vorbis, Musepack, AAC, ATRAC and Windows Media Audio Lossy (WMA lossy).

- Uncompressed audio format

One major uncompressed audio format, LPCM, is the same variety of PCM as used in Compact Disc Digital Audio and is the format most commonly accepted by low level audio APIs and D/A converter hardware. Although LPCM can be stored on a computer as a raw audio format, it is usually stored in a .wav file on Windows or in a .aiff file on Mac OS. The AIFF format is based on the Interchange File Format (IFF), and the WAV format is based on the similar Resource Interchange File Format (RIFF). WAV and AIFF are not inherently lossless; they're designed to store a wide variety of audio formats, lossless and lossy; they just add a small, metadata-containing header before the audio data to declare the format of the audio data, such as LPCM with a particular sample rate, bit depth, number of channels. Since WAV and AIFF are widely supported and can store LPCM,

- Lossless compressed audio format

A lossless compressed format stores data in less space without losing any information. The original, uncompressed data can be recreated from the compressed version.

Uncompressed audio formats encode both sound and silence with the same number of bits per unit of time. Encoding an uncompressed minute of absolute silence produces a file of the same size as encoding an uncompressed minute of music. In a lossless compressed format, however, the music would occupy a smaller file than an uncompressed format and the silence would take up almost no space at all.

- Lossy compressed audio format

Lossy compression enables even greater reductions in file size by removing some of the audio information and simplifying the data. This of course results in a reduction in audio quality, but a variety of techniques are used, mainly by exploiting psychoacoustics, to remove the parts of the sound that have the least effect on perceived quality, and to minimize the amount of audible noise added during the process. The popular MP3 format is probably the best-known example, but the AAC format found on the iTunes Music Store is also common. Most formats offer a range of degrees of compression, generally measured in bit rate. The lower the rate, the smaller the file and the more significant the quality loss.

(b) Chroma key-

Chroma key compositing, or chroma keying, is a special effects / post-production technique for compositing two images or video streams together based on color hues. The technique has been used heavily in many fields to remove a background from the subject of a photo or video – particularly the news-casting, motion picture and videogame industries. A color range in the top layer is made transparent, revealing another image behind. The chroma keying technique is commonly used in video production and post-production. This technique is also referred to as color keying, colour-separation overlay primarily by the BBC, or by various terms for specific color-related variants such as green screen, and blue screen – chroma keying can be done with backgrounds of any color that are uniform and distinct, but green and blue backgrounds are more commonly used because they differ most distinctly in hue from most human skin colors. No part of the subject being filmed or photographed may duplicate a color used in the background.

It is commonly used for weather forecast broadcasts, wherein a news presenter is usually seen standing in front of a large CGI map during live television newscasts, though in actuality it is a large blue or green background. When using a blue screen, different weather maps are added on the parts of the image where the color is blue. If the news presenter wears blue clothes, his or her clothes will also be replaced with the background video. A complementary system is used for green screens. Chroma keying is also used in the entertainment industry for special effects in movies and videogames. The advanced state of the technology and much commercially available computer software, such as Autodesk Smoke, Final Cut Pro, Pinnacle Studio, Adobe After Effects, and dozens of other computer programs, makes it possible and relatively easy for the average home computer user to create videos using the "chromakey" function with easily affordable green screen or blue screen kits.

(c) 3D graphics pipeline –

In 3D computer graphics, the graphics pipeline or rendering pipeline refers to the sequence of steps used to create a 2D raster representation of a 3D scene.[1]Plainly speaking, once a 3D model has been created, for instance in a video game or any other 3D computer animation, the graphics pipeline is the process of turning that 3D model into what the computer displays.[2] In the early history of 3D computer graphics, fixed purpose hardware was used to speed up the steps of the pipeline through a fixed-function pipeline. Later, the hardware evolved, becoming more general purpose, allowing

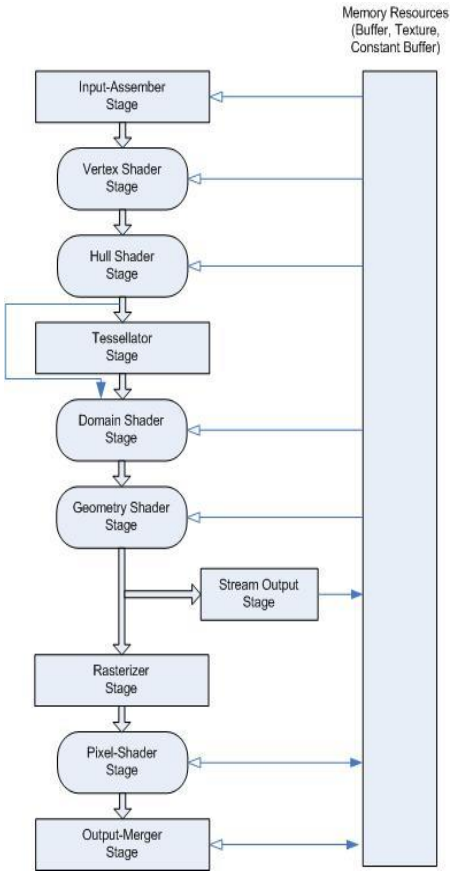
greater flexibility in graphics rendering as well as more generalized hardware, and allowing the same generalized hardware to perform not only different steps of the pipeline, like in fixed purpose hardware, but even in limited forms of general purpose computing. As the hardware evolved, so did the graphics pipelines, the OpenGL, and DirectX pipelines, but the general concept of the pipeline remains the same.

The 3D pipeline usually refers to the most common form of computer 3D rendering, 3D polygon rendering, distinct from raytracing, and raycasting. In particular, 3D polygon rendering is similar to raycasting. In raycasting, a ray originates at the point where the camera resides, if that ray hits a surface, then the color and lighting of the point on the surface where the ray hit is calculated. In 3D polygon rendering the reverse happens, the area that is in view of the camera is calculated, and then rays are created from every part of every surface in view of the camera and traced back to the camera.

sequence operations handles only simple primitives by design point, lines, triangles, quads (as two triangles) efficient algorithm complex primitives by tessellation complex curves: tessellate into line strips complex surfaces: tessellate into triangle meshes "pipeline" name derives from architecture design sequences of stages with defined input/output easy-to-optimize, modular design.

Vertex processing input: vertex data (position, normal, color, etc.) output: transformed vertices in homogeneous canonical view-volume, colors, etc. applies transformation from object-space to clip-space passes along material and shading data clipping and rasterization turns sets of vertices into primitives and fills them in output: set of fragments with interpolated data.

Fragment processing output: final color and depth traditionally mostly for texture lookups lighting was computed for each vertex today, computes lighting per-pixel framebuffer processing output: final picture hidden surface elimination compositing via alpha-blending.



The following list specifies the purpose of each of the stages.

- Input-Assembler Stage - The input-assembler stage supplies data (triangles, lines and points) to the pipeline.
- Vertex-Shader Stage - The vertex-shader stage processes vertices, typically performing operations such as transformations, skinning, and lighting. A vertex shader always takes a single input vertex and produces a single output vertex.
- Geometry-Shader Stage - The geometry-shader stage processes entire primitives. Its input is a full primitive (which is three vertices for a triangle, two vertices for a line, or a single vertex for a point). In addition, each primitive can also include the vertex data for any edge-adjacent primitives. This could include at most an additional three vertices for a triangle or an additional two vertices for a line. The geometry shader also supports limited geometry amplification and de-amplification. Given an input primitive, the geometry shader can discard the primitive, or emit one or more new primitives.
- Stream-Output Stage - The stream-output stage streams primitive data from the pipeline to memory on its way to the rasterizer. Data can be streamed out and/or passed into the rasterizer. Data

streamed out to memory can be recirculated back into the pipeline as input data or read-back from the CPU.

- Rasterizer Stage - The rasterizer clips primitives, prepares primitives for the pixel shader, and determines how to invoke pixel shaders.
- Pixel-Shader Stage - The pixel-shader stage receives interpolated data for a primitive and generates per-pixel data such as color.
- Output-Merger Stage - The output-merger stage combines various types of output data (pixel shader values, depth and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.
- Hull-shader, tessellator, and domain-shader stages, which comprise the tessellation stages - The tessellation stages convert higher-order surfaces to triangles for rendering within the Direct3D 11 pipeline.

e) sprites

CSS Sprites are a means of combining multiple images into a single image file for use on a website, to help with performance.

Sprite may seem like a bit of a misnomer considering that you're create a large image as opposed to working with many small ones, but the history of sprites, dating back to 1975, should help clear things up.

To summarize: the term "sprites" comes from a technique in computer graphics, most often used in video games. The idea was that the computer could fetch a graphic into memory, and then only display parts of that image at a time, which was faster than having to continually fetch new images. The sprite was the big combined graphic.

CSS Sprites is pretty much the exact same theory: get the image once, and shift it around and only display parts of it. This reduces the overhead of having to fetch multiple images.

It may seem counterintuitive to cram smaller images into a larger image. Wouldn't larger images take longer to load?

Let's look at some numbers on an actual example:

Image	File Size	Dimensions
canada.png	1.95KB	256 x 128
usa.png	3.74KB	256 x 135

Image	File Size	Dimensions
mexico.png	8.69KB	256 x 147

That adds up to a total of 14.38KB to load the three images. Putting the three images into a single file weighs in at 16.1KB. The sprite ends up being 1.72KB larger than the three separate images. This isn't a big difference, but there needs to be a good reason to accept this larger file... and there is!

While the total image size (sometimes) goes up with sprites, several images are loaded with a single HTTP request. Browsers limit the number of concurrent requests a site can make and HTTP requests require a bit of handshaking.

Thus, sprites are important for the same reasons that minifying and concatinating CSS and JavaScript are important.

GAMING ARCHITECTURE AND PROGRAMMING (GAP) MAY 2013 INFORMATION TECHNOLOGY

SEMESTER 8

Con.8226-13. **(REVISED COURSE)** **GS-3286**

(3 Hours) **[Total Marks : 100]**

N.B.: (1) Question No.1 is compulsory.

(2) Attempt any four questions out of remaining six questions.

1. Explain the following in detail(any two) :- [20 Marks]

(a) Implicit Invocation

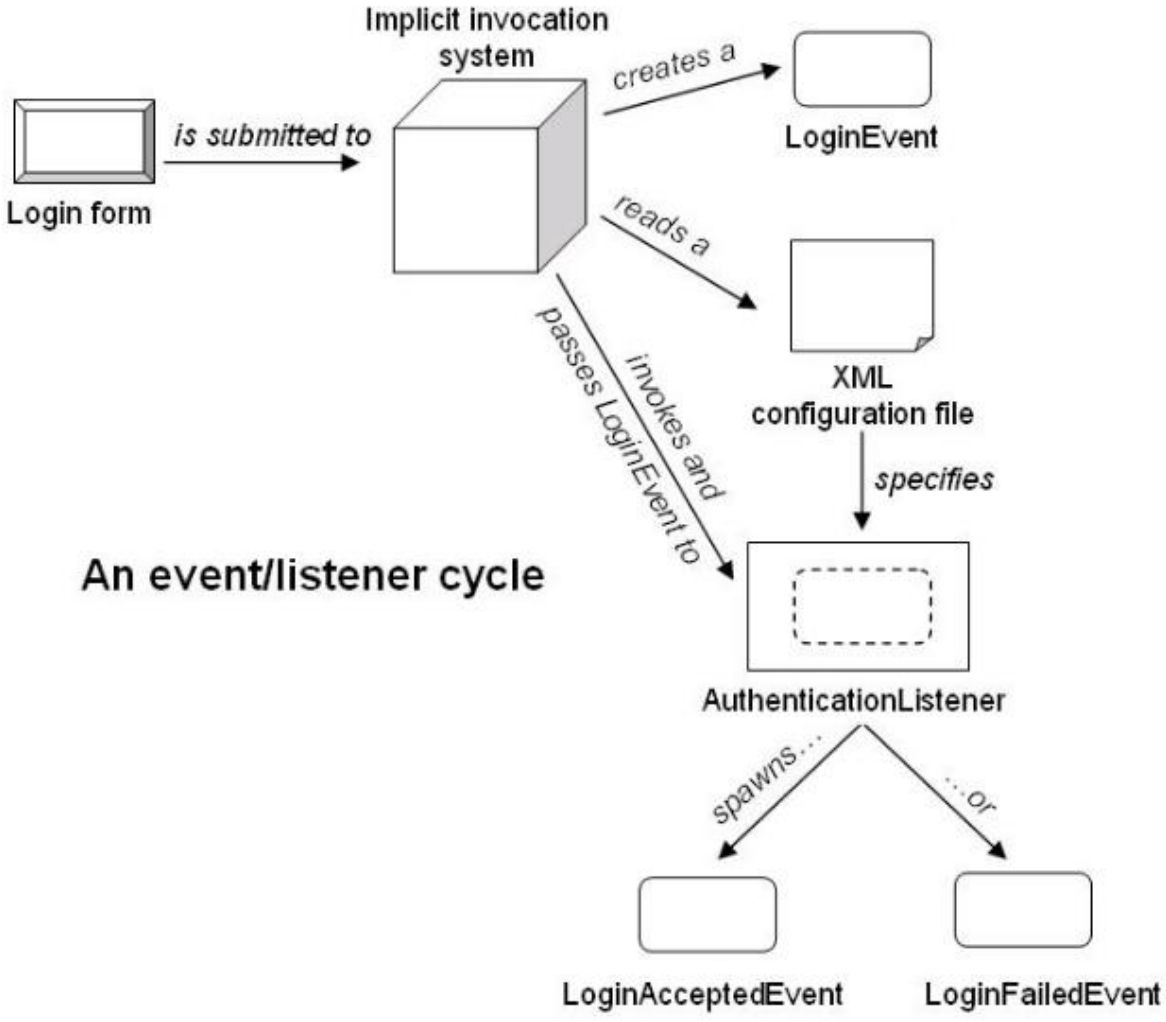
Event based, implicit invocation is an example of a well-crafted architectural style with high cohesion and loose coupling. As such, it is one of the more broadly accepted architectural styles in software engineering. Examples of implicit invocation systems abound, including virtually all modern operating systems, integrated development environments, and database management systems.

Garland and Shaw describe implicit invocation systems: "The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event 'implicitly' causes the invocation of procedures in other modules."

Implicit invocation systems are driven by events. Events are triggered whenever the system needs to do something—such as respond to an incoming request. Events can take many forms across different types of implementations; often for object-based systems an event is an object whose properties contain any contextual information needed to process the event (similar to how a HTTP request carries with it all its form and query-string variables).When an event is announced, the system looks up listener components for that event. Listeners fit the same criteria for components that we've already discussed—they are functional modules

of the system.

Components that wish to act as listeners are registered to listen for certain events at configuration time (by specification in an XML file, for instance). When an event is triggered, all registered listeners of that event are passed the event by means of a dynamically- determined method call. In this way, functions are implicitly invoked. This process of notifying listeners of an event is called event announcement



Events and listeners can themselves trigger other events. Let's consider a how a common Login authentication scenario can be represented in terms of events and listeners. In this example, a login form is filled out by a user and the form submitted. The incoming HTTP request triggers the creation of a LoginEvent, and the system populates the event with information in the request.

Next, the system determines the listeners for LoginEvent; in this case there is only one —the AuthenticationListener. Determined by a configuration file, the system invokes the AuthenticationListener's tryLogin() method, passing to it the event. Based on information in the event, the tryLogin() method will seek to authenticate the user. If the authentication succeeds, a new LoginAcceptedEvent is triggered. If authentication fails, a new LoginFailedEvent is triggered. The cyc le then continues, with any listeners of the new event being notified.

Implicit invocation architectures differ from explicit invocation systems in that implicit invocation system components use events to communicate with each other. Connectors in such architectures are bindings between events and component methods. Because these bindings are determined dynamically at runtime, components are loosely coupled; there is no compile-time determination of which method calls will be made. Loose coupling offers software architects the great benefit of increased flexibility and maintainability: new components can be added by simply registering them as event listeners.

Loosely coupled components work together, but do not rely on each other to do their own job. The interaction policy is separate from the interacting components, providing flexibility. Components can be introduced into a system simply by registering them for events of the system, aiding greatly in reusability. Introduction of new components does not require change in other component interfaces, providing scalability as new features are added. Overall, implicit invocation eases system evolution.

(b) Object factory

The object factory is a class whose sole purpose is to allow the creation of families of objects. This means that the code creating the objects is not tied in specifically to the objects that it is creating. Usually, all the objects created by the factory derive from the same abstract base class, and are returned to the requesting client as a reference to this class. You could also have multiple methods, Make XXXX(), each returning a different base class, but personally we prefer to have one factory per object family.

For example, in *Balls!*, all of the tokens are created by an object factory. Among other things, this makes the loading of levels very simple. Each token type has a unique ID, and loading a level is a

simple matter of passing that ID to the object factory, and making use of the pointer returned. One of the other advantages of this is low-cost garbage collection and memory tracking.

The object factory can keep track of all the objects that it allocates, inserting them into a list. When these objects are deleted, they remove themselves from this list. If any of these objects have not been freed at the end of the level, then they can either be cleaned up automatically or reported as errors. This helped us find quite a few insidious little bugs in the code.

Figure shows the class relationships. This describes how the objects are related to each other programmatically. In Figure “game object factory” receives requests from clients to create a specific “game object.” This factory class is responsible for creating all objects representing tokens within a level, and keeps track of them by using a related class, the “memory tracker” class. The factory class produces instantiations of the requested object, which are returned to the client as a pointer to the base class of all in-game token classes, the game object class.

The game object class contains a memory tracker class as an instantiated member variable. When a game object class is instantiated, the memory tracker member is also instantiated as part of it. When instantiated, the constructor of this class stores a reference to its owner (the enclosing game object), and inserts itself into the list within the factory class.

When the enclosing game object is deleted, the destructor of the member memory tracker is invoked, which removes the reference to itself from the list within the factory class. When the factory object is itself destroyed, it checks through the list to see if any remaining game objects have not been deleted. These can be either freed automatically, or (preferably) reported as errors, to be tracked down later.

This sort of dynamic object creation is very flexible: With judicious use of the object factory, the structure of whole applications can be determined at runtime. As long as all the objects returned by a specific factory conform to the same interface and behavioral contracts, then it is possible to (for example) configure levels dynamically with fairly simple code, allow customization of the user interface, release game expansions that don’t need to modify the original game code, or accomplish any number of tasks that would be tricky by other means.

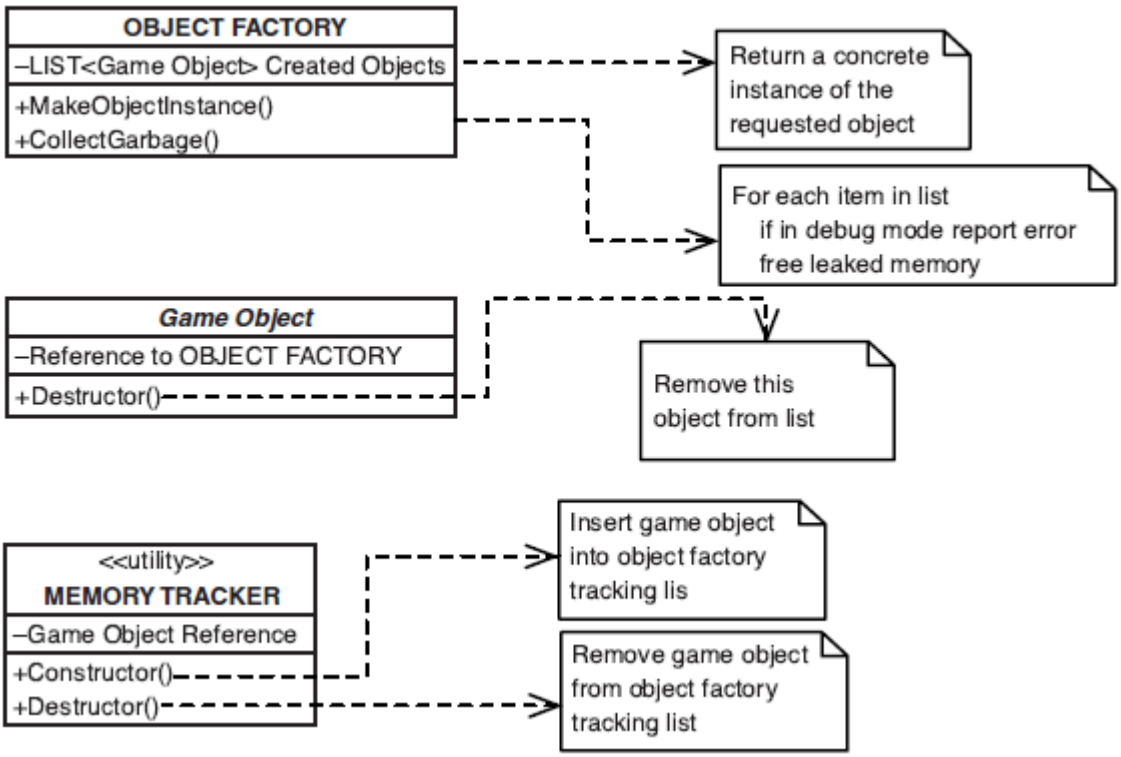


Figure : class relationships for the game object factory

(c) Architectural style

It is useful to organize human activity in games, but buildings aren't the most efficient way to do it. There's no real need to visit a building called the "Town Hall" in an online game when you could just send email to whoever works there; but the building provides a convenient metaphor for the functions that the Town Hall provides. Theft, likewise, may or may not be possible in games; if it is possible, a building provides a convenient metaphor for concealment and protection -- a way of indicating that an item is inaccessible to thieves.

In Age of Empires, once a resource is placed in the storage pit, it's protected from theft. The storage pit is really a magic place that converts resources from being vulnerable and unusable, to invulnerable and available for consumption. The game could call it anything it likes, but it calls it a building. It's not much like a real building, though: it never fills up, and if you burn it down you don't lose the contents. The Treasury in *Dungeon Keeper* was more like a real treasury: it could get full, and people could steal money out of it if it wasn't guarded.

Two functions that do translate over directly are military activity and general decoration. Just about all wargames make use of constructed edifices as a means of concealment and protection for troops, and any game that tries to create a sense of place uses architecture to define how that place feels to be. In short, buildings in games mimic the real world when necessary or aesthetically desirable, but this is not always the case. There are no buildings in chess.

Games do have a problem portraying outdoor spaces. Because of the limitations of looking at a monitor, we can't create sweeping vistas or panoramas that feel like the real thing. If you've ever tried to photograph the emptiness of a desert or the Great Plains, you'll know what I mean: an essential part of the experience is the sense of being surrounded by vast open space. Players sitting in a room, looking at a CRT, never feel that way. Another part of that sensation comes from the sheer length of time it takes to get anywhere. Most games allow you to move pretty fast -- no more than a few minutes to walk from one side of the world to the other so, the sense of scale is diminished. And of course aerial perspectives reduce the impressiveness of everything: the Great Pyramid is no big deal from 5000 feet up.

We're not very good at natural objects, either. In 3D games, straight lines are cheap and curves are expensive, so we tend to avoid curves. But look at an oak leaf: it's nothing but curves. With thousands of leaves per tree and thousands of trees in a forest, there's a good reason why we leave forests alone. As a result, most 3D games tend to feel rather sparse and sterile. Bauhaus, yes; botanical gardens, no.

The primary function of architecture in games is to support the gameplay. Buildings in games are not analogous to buildings in the real world, because most of the time their real-world functions are either irrelevant -- the real-world activity that the building serves isn't meaningful in the game -- or purely metaphorical. Rather, buildings in games are analogous to movie sets: incomplete, false fronts whose function is to support the narrative of the movie. Movie sets create context and support suspension of disbelief. They also diverge from the real world for narrative purposes. Consider New York as seen in a movie by Woody Allen, who loves the place, versus New York as seen in Taxi Driver. Sets are part of the story; they can make a place seem more (or less) beautiful, dangerous, tacky, etc. than it really is.

Gameplay (in non-social games) consists of challenges and actions taken to overcome them; architecture supports the gameplay by helping to define the challenges. There are four major ways in which this happens: constraint, concealment, obstacles or tests of skill, and exploration.

Constraint: In board games like chess and checkers, there are no boundaries except for the edge of the board. The challenge of the game is created by the rather arbitrary rules governing how the pieces may move. In representational games, we want units to move the way they would in real life, not according to some artificial rule; but most of the time we don't want them moving anywhere they like. Architecture establishes boundaries that limit the freedom of movement of avatars or units. It also establishes constraints on the influence of weapons. As a general rule, projectiles do not pass through walls (no matter how flimsy) nor do explosions knock them down, nor fires burn through them.

Concealment: Few computer games are games of perfect information, in which the player knows everything there is to know about the state of the game. Architecture is used to hide valuable (and sometimes dangerous) objects from the player; it's also used to conceal the players from one another, or from their enemies.

Obstacles and tests of skill: Chasms to jump across, cliffs to climb, trapdoors to avoid -- all these are part of the peculiar landscape architecture of computer games. Some of them can be surmounted by observation and logic, others by hand-eye coordination.

Exploration: Not quite the same as overcoming obstacles, exploration challenges the player to understand the shape of the space he's moving through, to know what leads to where. Mazes are of course one of the oldest examples of such a challenge. If the game doesn't give the player a map, he may have to rely on his memory to learn his way around. In recent years we have started making better use of subtle clues: sunlight coming through a window means that we're near the outside; a differently-shaded patch of wall indicates a secret door.

Persistent worlds like *Everquest* use buildings for a variety of social functions as well, of course, but as those are largely obvious and symbolic, I won't address them here.

Some time ago I came across the website of Canadian game designer Peter Lo). Included on his site was a sketch of a long ventilation shaft leading from the roof of a building straight down into an equipment room on the ground floor. The sketch included the following notation:

Considered as real-world architecture, this is isn't very sensible. The fans must blow out rather than in (that's why you don't plummet if you jump in while they're on). You might need two fans in order to move a given volume of air, but why would you need two sets of shutters? And why in the world are the shutters electrified? Above all, the remainder of the building is undefined. Like a movie set, it's just a false front, a container for the ventilation shaft and the equipment room below.

As game design, however, it's perfectly functional, though not entirely obvious to the inexperienced gamer. It supplies constraint (the player starts on the roof and must go down the ventilation shaft to get to the equipment room); an obstacle challenge (the fan blades and electrified shutters which, reading between the lines, we can tell must go on and off at intervals); and an exploration challenge (the player doesn't know what's at the bottom of the shaft until she gets there).

2. (a) Explain different types of game genre with an example. [10 Marks]

A video game genre is a specific category of games related by a common gameplay characteristic. Genres are not usually defined by the actual content of the game or its medium of play, but by its common challenge.[1]

Genres may encompass a wide variety of games, leading to even more specific classifications called *subgenres*. For example, an action game can be classified into many subgenres such as platform games and fighting games. Some games, most notably browser and mobile games, are commonly classified into multiple genres.

The following is a list of all commonly-defined video game genres, with short descriptions for individual genres

Action games emphasize physical challenges that require eye-hand coordination and motor skill to overcome. They center around the player, who is in control of most of the action. Most of the earliest video games were considered action games; today, it is still a vast genre covering all games that involve physical challenges.

Action games are classified many subgenres. Platform games and fighting games are among the best-known subgenres, while shooter games became and continue to be one of the dominant genres in video gaming since the 1990s. Action games usually involve elements of twitch gameplay.

- Platform games

Platform games are set in an environment with platforms, hence the name *platform game*.Platform games (or *platformers*) are set in a vertical or three-dimensional (3D) environment. Players guide a character through obstacles, jumping on platforms and battling enemies in order to advance. They often involve unrealistic physics and special movement abilities.

Donkey Kong was one of the earliest and best-known platformers; the American gaming press classified it using the term *climbing game* at the time *Super Mario Bros.* was one of the best-selling games of all time; more than 40 million copies were sold (excluding Game Boy Advance and Virtual Console sales). *Jumping Flash!* introduced 3D graphics to the genre, being the first console platformer to incorporate 3D graphics.

- Action-adventure

Action-adventure games combine elements of their two component genres, typically featuring long-term obstacles that must be overcome using a tool or item as leverage (which is collected earlier), as well as many smaller obstacles almost constantly in the way, that require elements of action games to overcome. Action-adventure games tend to focus on exploration and usually involve item gathering, simple puzzle solving, and combat. "Action-adventure" has become a label which is sometimes attached to games which do not fit neatly into another well known genre.

The first action-adventure game was the Atari 2600 game *Adventure* (1979). It was directly inspired by the original text adventure, *Colossal Cave Adventure*. In the process of adapting a text game to a console with only a joystick for control, designer Warren Robinett created a new genre. Another typical Action-Adventure game is "The Legend of Zelda" by Nintendo, which involves puzzle solving, sword fighting, and item collecting. Because of their prevalence on video game consoles and the absence of typical adventure games, action-adventure games are often called "adventure games" by modern gamers.

- Stealth game

Stealth games are a somewhat recent subgenre, sometimes referred to as "sneakers" or "creepers" to contrast with the action-oriented "shooter" subgenre. These games tend to emphasize subterfuge and precision strikes over the more overt mayhem of shooters, for example, the Sly Cooper series.

- Adventure

Adventure games were some of the earliest games created, beginning with the text adventure *Colossal Cave Adventure* in the 1970s. That game was originally titled simply "Adventure," and is the namesake of the genre. Over time, graphics have been introduced to the genre and the interface has evolved.

Unlike adventure films, adventure games are not defined by story or content. Rather, adventure describes a manner of gameplay without reflex challenges or action. They normally require the player to solve various puzzles by interacting with people or the environment, most often in a non-confrontational way. It is considered a "purist" genre and tends to exclude anything which includes action elements beyond a mini game.

Because they put little pressure on the player in the form of action-based challenges or time constraints, adventure games have had the unique ability to appeal to people who do not normally play video games. The genre peaked in popularity with the 1993 release of *Myst*, the best-selling PC game of all time up to that point. It had four proper sequels, but none managed to experience the same level of success. The success of *Myst* also inspired many others to create similar games with first person perspectives, surreal environments and minimal or no dialogue, but these neither recaptured the success of *Myst* nor of earlier personality-driven adventures.

- Role-playing video games draw their gameplay from traditional role-playing games like *Dungeons & Dragons*. Most of these games cast the player in the role of one or more "adventurers" who specialize in specific skill sets (such as melee combat or casting magic spells) while progressing through a predetermined storyline. Many involve manoeuvring these character(s) through an overworld, usually populated with monsters, that allows access to more important game locations, such as towns, dungeons, and castles.

Since the emergence of affordable home computers coincided with the popularity of paper and pencil role-playing games, this genre was one of the first in video games and continues to be popular today. Gameplay elements strongly associated with RPGs, such as statistical character development through the acquisition of experience points, have been widely adapted to other genres such as action-adventure games.

Though nearly all of the early entries in the genre were turn-based games, many modern role-playing games progress in real-time. Thus, the genre has followed the strategy game's trend of moving from turn-based to real-time combat.

(b) Discuss the contents of game design document. [10 Marks]

A game design document may be made of text, images, diagrams, concept art, Concept art is a form of illustration used to convey an idea for use in films, video games, animation, comic books or other media before it is put into the final product.[1] Concept art is also referred to as visual development and/or concept design. This term can also be applied to retail, set, fashion, architectural and industrial design.

Concept art is developed in several iterations. Artists try several designs to achieve the desired result for the work, or sometimes searching for an interesting result. Designs are filtered and refined in stages to narrow down the options. Concept art is not only used to develop the work, but also to show the project's progress to directors, clients and investors. Once the development of the work is complete, advertising materials often resemble concept art, although these are typically made specifically for this purposed, based on final work or any applicable media to better illustrate design decisions.

Some design documents may include functional prototypes Software prototyping is the activity of creating prototypes of software applications, i.e., incomplete versions of the software program being developed. It is an activity that can occur in software development and is comparable to prototyping as known from other fields, such as mechanical engineering or manufacturing.

A prototype typically simulates only a few aspects of, and may be completely different from, the final product.

Prototyping has several benefits: The software designer and implementer can get valuable feedback from the users early in the project. The client and the contractor can compare if the software made matches the software specification, according to which the software program is built. It also allows the software engineer some insight into the accuracy of initial project estimates and whether the deadlines and milestones proposed can be successfully met. The degree of completeness and the

techniques used in the prototyping have been in development and debate since its proposal in the early 1970s or a chosen game engine

A game engine is a software framework designed for the creation and development of video games. Developers use them to create games for consoles, mobile devices and personal computers. The core functionality typically provided by a game engine includes a rendering engine ("renderer") for 2D or 3D graphics, a physics engine or collision detection (and collision response), sound, scripting, animation, artificial intelligence, networking, streaming, memory management, threading, localization support, and a scene graph. The process of game development is often economized, in large part, by reusing/adapting the same game engine to create different games, or to make it easier to "port" games to multiple platforms for some sections of the game.

Although considered a requirement by many companies, a GDD has no set industry standard form. For example, developers may choose to keep the document as a word processed document, Concept art is a form of illustration used to convey an idea for use in films, video games, animation, comic books or other media before it is put into the final product. Concept art is also referred to as visual development and/or concept design. This term can also be applied to retail, set, fashion, architectural and industrial design.

Concept art is developed in several iterations. Artists try several designs to achieve the desired result for the work, or sometimes searching for an interesting result. Designs are filtered and refined in stages to narrow down the options. Concept art is not only used to develop the work, but also to show the project's progress to directors, clients and investors. Once the development of the work is complete, advertising materials often resemble concept art, although these are typically made specifically for this purposed, based on final work or as an on-line collaboration tool. Collaborative software or groupware is an application software designed to help people involved in a common task to achieve their goals. One of the earliest definitions of collaborative software is 'intentional group processes plus software to support them.

Collaborative software is a broad concept that overlaps considerably with computer-supported cooperative work (CSCW) groupware is part of CSCW. The authors claim that CSCW, and thereby groupware, addresses "how collaborative activities and their coordination can be supported by means of computer systems." Software products such as email, calendaring, text chat, wiki, and bookmarking belong to this category whenever used for group work, whereas the more general term social software

applies to systems used outside the workplace, for example, online dating services and social networking sites like Twitter and Facebook.

The use of collaborative software in the work space creates a collaborative working environment (CWE).

Finally, collaborative software relates to the notion of collaborative work systems, which are conceived as any form of human organization that emerges any time that collaboration takes place, whether it is formal or informal, intentional or unintentional. Whereas the groupware or collaborative software pertains to the technological elements of computer-supported cooperative work, collaborative work systems become a useful analytical tool to understand the behavioral and organizational variables that are associated to the broader concept of CSCW.

3. (a) What do you understand by blue sky research? Why is it dangerous? [10 Marks]

We would all love to spend all day in front of a computer munching pizza and guzzling coffee, freely engaging our creativity and researching whatever we like. Some companies do allow this and have active (if somewhat variable) research departments. That's a good thing. Research is essential to the survival of a company. It's *blue-sky* research that you have to be careful about. By definition, any technology present in games currently available in the shops is about 18 months behind what is currently being worked on by the best design houses.

In other words, even with the best will in the world, it is statistically unlikely that anything remotely useful will be achieved, except a rather tragic comedy of errors. When a game project depends on the outcome of research that has not been completed, that project is in great danger. Putting anything in the critical path for which the outcome cannot be predicted is sheer idiocy; but for some reason—whether greed, stupidity, or just plain ignorance—development teams seem to do this on a regular basis.

We're not advocating that research should be abolished; that would be a draconian measure and would lead to further stagnation of the industry. What we *are* saying is that there is a time for research and a time for development, and that the two should never overlap. Any research that is instigated should be directed research. It should have an aim.

An example would be the design and development of a library useful in fuzzy-logic calculations. For sure, there would be a fair amount of research involved, but this would be research into ways of optimizing and improving on known techniques. The blue-sky alternative would be to look for a completely new method of doing things. Fortunately, there is very little you can imagine doing with a computer that hasn't been already done by someone somewhere.

If you're very lucky, then they have published their results. Building on the work of others, although less glamorous, is a surer way of getting good results. Besides, if you wanted glamour then why did you become a developer? When your team is researching, set a strict time limit and stick to it ruthlessly. Run with what you physically have at the end of the research period, not with what you are promised at the beginning.

The time allocated for research should also factor in the time necessary to bring the research project to a successful conclusion and with a stable and well-documented component. As the research progresses, more and more unknowns will become quantifiable. As soon as the fundamental technique being researched is working, a mini-schedule can be drawn up to allow for this tidying-up procedure. If the game design depends on this research, then this is the single most critical point of the whole development.

The project will succeed or fail—right here, right now—dependent on the success of this research project. Obviously, this is not a good idea: Gambling the whole project on blue-sky research is a game that only the very foolhardy or the extremely desperate would play. Remember, any safety net is better than no safety net. If no safety net is possible, then "make-or-break" research should be avoided. It's too much of a risk, no matter how cool the developer doing the research is.

Under most circumstances, releasing the product is better than canning it. Only the most cash-rich companies will be able to afford internal research such as that used during the development processes of *Die by the Sword* and *Outcast.* Not so many companies are able to afford the protracted development times and the risks that are associated with research of this nature. If your company doesn't have the sort of cash available to finance a research department, you still have a few options open.

The first, and most undesirable, is to avoid projects that require R&D. This is only really acceptable for the "ticking-over-and-we-know-we'll-only-sell-a-couple-of hundred- thousand-or-so" type of company. This is a viable option, but it isn't going to go anywhere fast.

For the less-affluent companies, another option is to form links with academia. Forming a loose relationship with universities and their computing departments can provide some good technology at some very good prices. A company that we have worked for in Belgium used their local university as a source of cutting-edge cryptography algorithms and regularly tested their latest algorithm designs by letting the university researchers loose on them. The theory was that any implementations of algorithms that these university cryptography experts couldn't crack were pretty much guaranteed to be secure.

If you use the "external research" system, then this also simplifies some other company decisions. Form a liaison with academia: A good relationship means that they get to publish and you get to use their new ideas. Yet another option is to be part of a large conglomerate, either using the Hollywood studio model or the loose alliance model typified by Lionhead's satellite. The latter may not be an optimal system.

To optimize it you need to ensure that the separate development teams are aware of and able to share each other's technology. Also you would ideally organize collateral R&D so that two companies don't waste time on the same tasks, all of which is best achieved by appointing a resource investigation unit to interface between all the projects in development. A loose development alliance can work, therefore, but it needs structure.

(b) What are the various phases in game development? State the process, people involved and the outcome of each phase. [10 Marks]

- Step One: Initial Planning

The Genesis Gaming Design and Marketing teams will meet with the client to determine the key concepts driving the development of a strategic game portfolio. This will address issues such as analysis of an existing portfolio, current demographics, additional player acquisition and retention, emerging trends and profiles. We will further discuss a range of themes, volatility levels, features, bonus games and any other aspect required for bespoke development of a strategic portfolio. It is also important that there is an overall marketing discussion to determine how the games can be used to further extend the client's brand.

- Step Two: Technical Review

Our Engineering Team will review documentation provided by the client to determine if there are any technical questions or matters that need to be addressed to best develop to a specific platform.This will ultimately save significant time in the integration process.

- Step Three: Initial Themes & Concept Art

We will submit themes, names, concept art and descriptions for consideration and approval or modification. We will then work with the client to determine portfolio development priorities. This allows the ability to set expectations as to game delivery. The client can then implement a schedule for marketing and release.

- Step Four: Features and Mathematics

Our Game Design and Mathematics departments will determine and create unique math models that best represent the client's objectives as stated through the analytical process and the client's objectives. Math will be verified and all required percentages will be provided.

- Step Five: Art and Creative Design

The Genesis Gaming Art Department will design and submit static art and design elements for approval. Subsequently, our Animation and Music Composition teams will finalize the game. This will require the client to provide appropriate documentation such as templates and other substantive game specifications for consistency.

Upon completion, our Demo team will provide a playable version of the game for additional review.

- Step Six: Integration

Upon approval, our Engineering Department will integrate the game to the appropriate provider platform. This requires the necessary documentation, such as an API, from the client along with server support, assuming it is not an "asset only" delivery. We place a significant emphasis on product assurance and quality control so the game will be very "client provider friendly."

- Step Seven: Delivery

All game assets, in the requested file formats, including all release documentation will be provided according to contractually specified delivery requirements. Genesis Gaming is also happy to provide any material that may be requested for the client's marketing campaign.Since success is our

mutual objective, it is our goal to make certain that the client has all tools possible to promote the best exclusive content in the industry

4. (a) Identify the tokens of any game of your choice and draw: [10 Marks]

(i) Token Interaction Matrix

Tokens do not interact with each other on their own. An occurrence of an event causes the interaction. A token interaction matrix is a chart of all interactions that take place in the game. It helps when deciding the object-oriented architecture. The interactions can be of the following types:

- Symmetric interaction: If the interaction is same in both the ways. Denoted as "-"
- Asymmetric Interaction: This interaction depends on the direction

Symmetric interactions are the same both ways. For example, the behavior of the ball and the bat are not different depending on how we consider the collision. That is, if we say that the ball collides with the bat, we would expect exactly the same results to occur as if we had said the bat collides with the ball. The semantics do not matter.Symmetric interactions are shown as squares in the matrix.

Asymmetric interactions are shown in the matrix as a square split into two triangles. An asymmetric interaction is different depending on the direction. In this case, the semantics do matter. Each triangle represents one direction of the interaction. Taking the solitary case from the *Pong* matrix as an example, we could say that a goal causes the score to increment by one, but the score incrementing by one does not cause a goal to occur—cause-effect, not effect-cause. (There'll be no breaking the laws of causality in this book!) The matrix allows us to perform a visual check on our interactions. We can check that they are what we would expect, and we can see if we have missed any or made any errors. We may be able to spot unexpected chain reactions (the sort of things that in some cases will enhance a game, but in other cases will render it virtually unplayable). These things will be picked up in play-testing, but the sooner it is spotted the cheaper it is to fix.Some interactions are not to be considered at all depending on Rules of game .Denoted by "X"

(ii)Token Class Hierarchy

In general, a token is an object that represents something else, such as another object (either physical or virtual), or an abstract concept as, for example, a gift is sometimes referred to as a token of the. We can also consider these tokens to be arranged in a form of hierarchical structure.

The playing area, or game world, in itself is at the top of the hierarchy. From then on in, it is an essentially flat hierarchy.

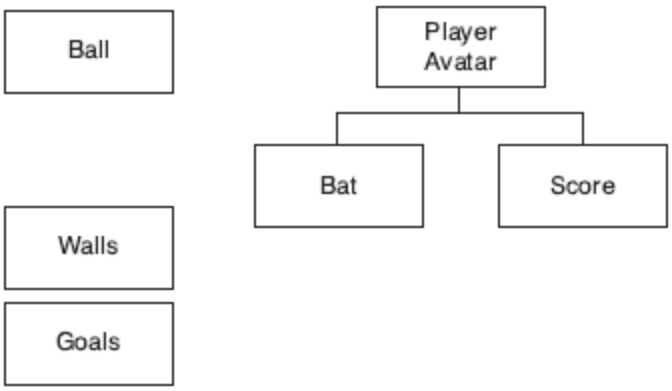


Figure :The *Pong* token hierarchy.

The game world token contains all the other tokens. Obviously every token has to operate within the game world in order to form a part of the game. The player avatar token is the representation of the player within the game world. It is effectively a channel for the user interface between the player and the game.

The player avatar for *Pong* is very simple; it is merely a bat and a score. These are how the player is represented in *Pong*. The other tokens—those manipulated by the computer—are the ball, the walls, and the goal zones.

Now it's time for a little sleight of hand of the sort that is possible with only the written word. Reread the two paragraphs, and for every instance of the word "token," read it as "object."

So, if we were just talking about objects all along, why didn't we just use the word "object" to start with?

The main reason—and why we particularly like the use of the word token and why we will continue using it from here on in—is that these conceptual tokens may not have a one-to-one mapping with the programming language objects (for example, in C++) that are defined by the programmers. What we are trying to do is to break down the game design into conceptual objects that will eventually

be translated into programming language objects. This tokenization process is an intermediary stage in the production of a decent architecture. In order to describe this without causing confusion, we need to use different terminology for each type of object.

The tokenization of a game design such as *Pong* is fairly trivial, and there's really only one way to do it. In spite of this, it makes an excellent example to try to demonstrate the thought processes behind tokenization. Not all games will be so trivial, and, for some more complex games, there may be many ways, all of which are equally valid. So now we have a set of tokens. On their own, they are not very exciting as they do not interact with one another. But as we know, in *Pong* there are all sorts of interactions going on. Well, one anyway: collisions.

We can now define an event—the collision event. Let's say that a collision event is generated when two tokens collide. The net result of this event is that each token receives a message telling it that a collision has occurred, and the type of object it has collided with.

The token interaction matrix is a very important construct. It is a chart of all the interactions that take place in the game. Note that for very large games we would not use a token-token matrix directly. Instead we would introduce an extra layer of abstraction by using token-property and/or property-property.

Okay, so let's look at the *Pong* token interaction matrix. The matrix is arranged in a triangular format, with each token listed along the side and the bottom. An unusual feature of *Pong* is that tokens do not come into contact with other tokens of the same type. This immediately means that the token-token interactions for bat-bat, ball-ball, wall-wall, goal-goal, and score-score can be discounted. Due to the nature of the game, the following interactions can also be discounted: bat-goal, wall-goal, score-bat, score-ball, and score-wall.

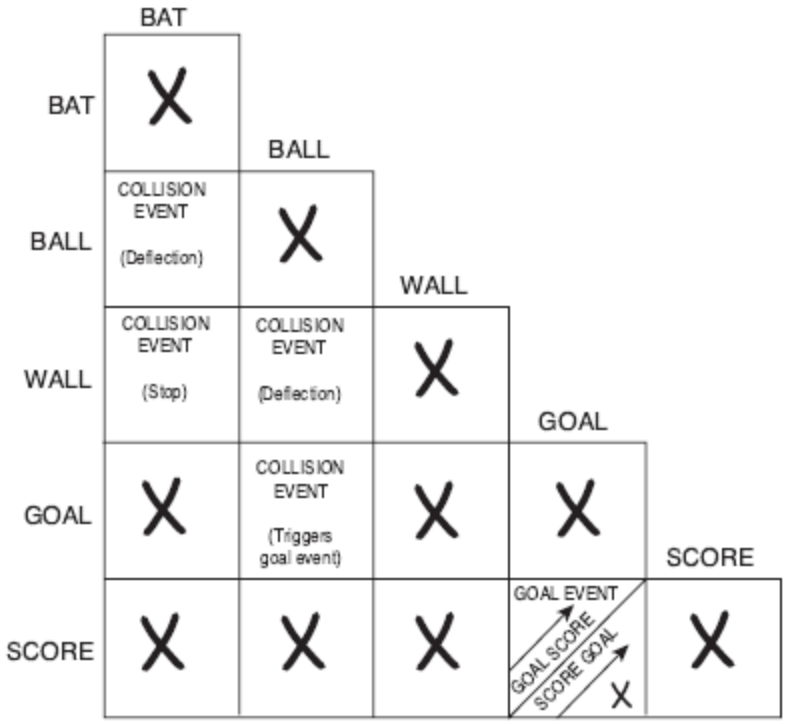


Figure : The *Pong* token interaction matrix.

(b) Mention the components in the main tiers of game architecture. [10 Marks]

One of the most effective development models for game development is the "waterfall," "spiral," or "tier-based" method."Development,"suffice it to say that these methods reduce the development into discrete tiers, with **each tier taking into account what was built before, and what will be built ahead.** Thus, we approach the architecture design in a holistic fashion. By this, we mean that, even as we split the architecture into a series of tiers, we still keep an eye on the whole architecture. When designing the architecture of a module, we first try to define the interface that will be required. The best starting approach is to build the skeleton of what we will need. Implementing this skeleton is the main aim of the first tier, and subsequent tiers concentrate on fleshing out the framework, filling in and replacing stubbed-out functionality with the real deal.

Tier Zero: The Prototype : Of course, it would be difficult to leap straight in and implement the skeleton correctly the first time you try, so we need to take this into account when designing the architecture. We do this via prototyping techniques. Prototyping is a

risk-reduction mechanism, allowing us to explore and evaluate simulated risks before we have to tackle them for real. We will be considering four main types in varying levels of detail.

- Gameplay prototype—The most important consideration!
- User-interface prototype—How does the game interact with the player?
- Subsystem prototypes—How do all the subsystems interact with each other? What interfaces do they export for use with other systems?
- Algorithm prototypes—What is the best algorithm to use for a given situation? Is it suitable to be abstracted behind an interface so that we can literally "plug and play" the algorithm?

In fact, **the prototype is really the very first part of defining the architecture**. We could say that the prototype is Tier Zero in the development model. Tier Zero is really a special case, before we get into the architecture proper. We need to be able to test our game design ideas, refine them, and work out what we need to do in order to efficiently implement our tier-based architecture. This tier of our project isn't really one of the main tiers. It's really a "pre-project" investigation. How are we going to put our architecture together? What will work? What doesn't? How does our game design hold up to interrogation under the harsh light of critical analysis?

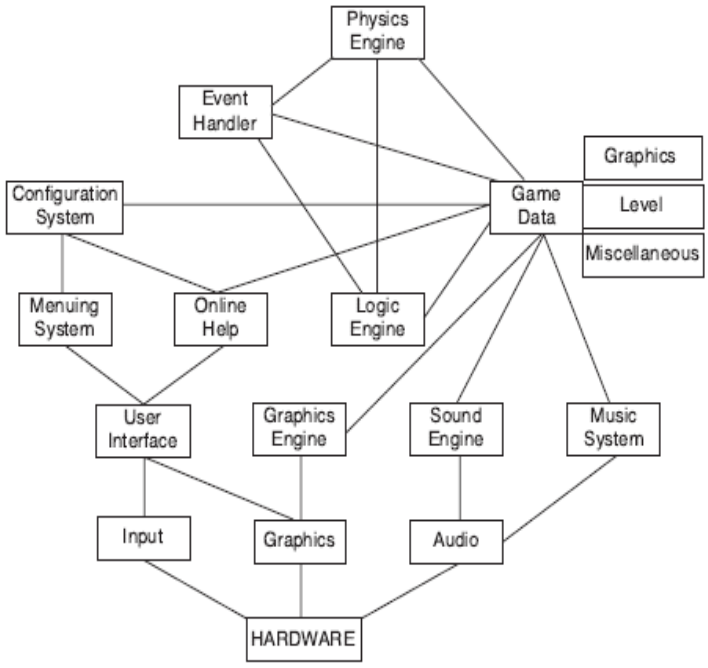


Figure 2.1 The architecture of game

When developing the components for the game (and we are still talking within the realms of the hard architecture here), then we should be drawing on the components developed by the software factory system.

When building the architecture, we should take into account whether we can make use of already existing components, whether they are in-house or not. Increasingly, it seems that the biggest factor in game development seems to be the time to market, so any use of existing components that can shave precious time off the schedule should be seriously considered.

The buzzwords that we need to keep in mind are *component-based architecture.* If we build our architecture up piece by piece, selecting each piece from a virtual catalog (as the Victorian English did when they built their houses from pattern books), then we should end up with a cleaner and more segmented architecture.

Even big players such as Sony have seen the benefits of this scheme—which has been dubbed *middleware—and* are leveraging those benefits by licensing third-party physics engines and other components to incorporate into their developer SDK for the PlayStation 2.

In the situations in which we have to develop a new component for our project, we should consider the potential for reuse. Most game platforms nowadays are powerful enough so that we simply do not need to put any of the game-specific code into hard architecture modules. There really is no excuse for missing reuse opportunities, because the major opportunities for reuse are going to be at the level of the hardware interface and general algorithm.

It's quite easy to imagine reusing a graphics engine or a sound engine, which are at the lowest level of granularity within our architecture, but with a little effort we can also reuse components from higher in the hierarchy. For example, with careful design we can produce a menuing system that can be used across a number of projects.

This is not such a strange concept—after all; a large part of Windows itself is taken up by a common user interface that can be used across projects. This does not mean that we'd want our games to conform to the Windows user interface because we would consider them boring, but the concept can be stolen and used to provide a simple configurable menu system that we could happily use across an entire range of projects without the customers even noticing that we had reused the code.

This is not the only opportunity within the architecture for reuse. You'll find plenty more as computers become more powerful and are able to take up the slack required to support the extended demands of a completely soft architecture.

5. (a) What are hard and soft architectures? Which one is preferred for easy maintenance and why? [10 Marks]

When we begin to consider the actual architecture of the game (refer figure 2.1), we need also to consider how to construct it around the planning needs of the schedule. "Milestones and Deadlines". Each milestone will specify the technical requirements to complete a particular tier. As we have been discussing, the architecture is specified in three main stages, each of which expands into a number of tiers.The three stages are:

- Prototyping : The prototyping stage allows us to have a dress rehearsal of the full architecture, allowing us to tackle any tricky points and difficulties that we might encounter. Of course, this doesn't mean that we are going to be able to cover all of these difficulties, but we can tackle at least the more obvious ones, and, of course, we will be able to explore gameplay issues sooner than we would be able to otherwise.

- Hard-architecture design : The hard-architecture design stage involves the laying of the game framework. However, the point of using a component- based design is to produce a set of generic components that can be used across projects. Hard-architecture components would need to be upgraded and augmented in order to keep up with emerging technology, but, with a sensible set of interfaces, the disruption caused by this continual upgrade (not replacement) process would be minimal. Only after a few projects have been undertaken will we see any benefits from reusing in-house components.Once a few projects have been completed, we will also be able to use our own components.

- Soft-architecture design : "The Software Factory," we mentioned that the software factory architecture caused an apparent drop in productivity for the first project developed using it, by applying morestructured development methods that reduce the amount of time spent on backtracking and unnecessary rework.This is pretty much unique for any project, and this is as it should be, for within the soft architecture is the unique spark that makes your game stand out from the rest. By using the soft-architecture system, we are taking advantage of all the groundwork that has already been done for us. The soft architecture defines the game-specific functionality and data required, such as in game graphics, music, and other data. In fact, the soft architecture is essentially data driven.

(b) Explain game play research. [10 Marks]

The gameplay could have an impact on the technology needed for the game. For example,the game's user interface may require investigation of certain types of controller. Other aspects of the gameplay may require active research. For example, in the case of strategy games, plenty of information is available if you're prepared to do a little digging: Game theory from the War Studies Group is readily available on the Web, as are analyses commissioned by the U.S. Navy and the Pentagon. Looking at

history also indicates where different factors have contributed to the payoff matrices of a real situation, such as why the Aztecs became so powerful, and how in spite of this, they were beaten by Cortés.

For researching the gameplay of a puzzle game like Tetris or Balls!, you could look at psychological work on types of reasoning. A satisfying puzzle game like Tetris or Puzzle Bobble will include spatial and temporal reasoning (the "story" part) as well as logical reasoning using the manipulation of abstract concepts mapped to concrete entities (the "planning" part) and the pleasurable payoff of watching where all that leads, and learning something more about the way the game rules operate (the "play" or "learning curve" part.)

Obviously, you can find more genres of games, but the point we are making here is that you don't necessarily have to look in the standard or clichéd places for gameplay ideas, such as novels, films, and other games. There is a whole world of information out there, and a lot of it can be applied to gameplay, even if it does not seem immediately obvious. After all, the idea for Tetris was drawn from the field of mathematics.

The last kind of research involves researching the technology that is required to actually implement the game. Are you going to be using any new techniques, or breaking into uncharted waters?

This sort of research is what id Software spends most of its time doing. id Software has lots of money. This is no coincidence: without lots of cash backing you up, research has to be more focused and specific. You can't just go wandering through your ideas randomly, idly daubing code "pan-it" on the "canvas" of your compiler without someone being willing to pay the bills.

The fact is that this sort of research takes a lot of time and a lot of money. This is where the real research is, and where the bulk of the budget of time and money will be invested. In an ideal world, your company would have enough money to allow unrestricted research into new technology.

Unfortunately, unless you are id Software, then this is very unlikely. There will always be commercial pressures breathing down your necks, and the company management will be expecting results. In previous chapters, we have pointed out the dangers of depending on concrete results from research. I'll not repeat those warnings here.

Research is an unpredictable activity, and research into new technologies is particularly difficult. For every Quake engine, there are probably hundreds of failed attempts. Worse still, a fair proportion of these failed attempts will have been for game projects that had to be canned due to that failure: For the original release of Quake, the focus was on the technology and not so much on the game.

In contrast, Quake II was much less of a technological advance over Quake than Quake was over Doom, A few refinements were made to the engine, but most of the effort appeared to have gone into refining the gameplay and the storyline. Quake III Arena seems to have reverted to the original Quake model: concentrating on the gameplay emerging from the technology (which has been improved by the addition of quadratic curve rendering).

This is an interesting approach that appears to defy conventional game theory. Even the single-player game is a multiplayer game, with all the other players controlled by the computer. I'd go as far to say that there is no real gameplay in a multiplayer game: It's more a simulation set in a fantasy environment. It's too close to reality (albeit a fictional reality) for it to be considered a game. It's an accurate simulation of future combat.

You may be getting the impression that we are against technology in some way. That's not true. We are against the gratuitous use of technology in the same way as we are against gratuitous violence. It's unnecessary and is, in some cases, quite disturbing.

The whole industry (with few exceptions) appears to be putting technology before gameplay, and this is a dismal and worrying prospect. We've lost count of the number of games we have seen that have lost gameplay value because the developers wanted to showcase their latest technologies. Case Studies 18.3 and 18.4 gives details of a much-welcome exception to this pattern.

Research should be treated with as much seriousness as you would find in a laboratory. Everything should be documented. Every thought, every procedure, and every result— even the wrong ones—needs to be recorded. This is serious stuff. Research is the lifeblood of your company. You need to research in order to keep up with the fast changing pace of technology. If not, you risk being left behind in the rush.

6. (a) What are Scripting languages and why are they preferred for game play? [10 Marks]

A glue language is a programming language (usually an interpreted scripting language) that is designed or suited for writing glue code – code to connect software components. They are especially useful for writing and maintaining:

- Custom commands for a command shell
- Smaller programs than those that are better implemented in a compiled language

"Wrapper" programs for executables, like a batch file that moves or manipulates files and does other things with the operating system before or after running an application like a word processor, spreadsheet, data base, assembler, compiler, etc.

- Scripts that may change
- Rapid prototypes of a solution eventually implemented in another, usually compiled, language.

- GUI scripting

With the advent of graphical user interfaces, a specialized kind of scripting language emerged for controlling a computer. These languages interact with the same graphic windows, menus, buttons, and so on that a human user would. They do this by simulating the actions of a user. These languages are typically used to automate user actions. Such languages are also called "macros" when control is through simulated key presses or mouse clicks, as well as tapping or pressing on a touch-activated screen.

These languages could in principle be used to control any GUI application; but, in practice their use is limited because their use needs support from the application and from the operating system. There are a few exceptions to this limitation. Some GUI scripting languages are based on recognizing graphical objects from their display screen pixels. These GUI scripting languages do not depend on support from the operating system or application.

- Application-specific languages

Many large application programs include an idiomatic scripting language tailored to the needs of the application user. Likewise, many computer game systems use a custom scripting language to express the programmed actions of non-player characters and the game environment. Languages of this sort are designed for a single application; and, while they may superficially resemble a specific general-purpose language (e.g. QuakeC, modeled after C), they have custom features that distinguish them. Emacs Lisp, while a fully formed and capable dialect of Lisp, contains many special features that make it most useful for extending the editing functions of Emacs. An application-specific scripting language can be viewed as a domain-specific programming language specialized to a single application.

- Extension/embeddable language

A number of languages have been designed for the purpose of replacing application-specific scripting languages by being embeddable in application programs. The application programmer (working in C or another systems language) includes "hooks" where the scripting language can control the application. These languages may be technically equivalent to an application-specific extension language but when an application embeds a "common" language, the user gets the advantage of being able to transfer skills from application to application. A more generic alternative is simply to provide a library (often a C library) that a general-purpose language can use to control the application, without modifying the language for the specific domain.

JavaScript began as and primarily still is a language for scripting inside web browsers; however, the standardization of the language as ECMAScript has made it popular as a general purpose embeddable language. In particular, the Mozilla implementation SpiderMonkey is embedded in several environments such as the Yahoo! Widget Engine. Other applications embedding ECMAScript implementations include the Adobe products Adobe Flash (ActionScript) and Adobe Acrobat (for scripting PDF files).

(b) What are principles to be followed for effective use of factory method? [10 Marks]

Factory Design patterns are Creational patterns and deals with how the objects are created. There are three design patterns, namely, Simple Factory, Factory Method and Abstract Factory design patterns. All these patterns hide the object creation process from the client and supply the objects without specifying the exact class of object that will be created. These patterns are widely used across programming languages especially Java and .Net. In this discussion, we will see the different factory pattern implementations and validate them against open-closed principle.

- The Open-Closed Principle

Open-Closed principle is one of the fundamental principle of object oriented programming and is a one of the principle of SOLID. According to Open-Closed Principle the "*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification* "There are many different views on Open-Closed Principle and its applicability, scope and practicality. Even some argues that, it is closely related to Protected Variation pattern of General Responsibility Assignment Software Patterns (GRASP) (Reference: Applying UML and Patterns Craig Larman). It says, protect elements from the variations of other elements by wrapping the focus of instability with an interface and using polymorphism to create various implementation of this interface.

- Factory Design Patterns and Open-Closed Principles

The question is, when you design your program or module using Factory design patterns, are you violating the Open-Closed principle?Let us check this by using sample programs of all the three Factory patterns.As we mentioned above, Factory patterns are used to implement the concept of factories and deals with the problem of creating objects (exact class of object that will be created will not be specified) also called products. Factory pattern comes in different variants and implementations such as GoF's Factory Method and Abstract Factory. First let us see the simple Factory pattern example. We will see the Factory Method and Abstract Factory patterns after this.

- Simple Factory Pattern

The simple Factory (method) pattern is really simple. The client requires a Product object and instead of creating the product using the new operator, it asks the Factory for a new object. The client also provides the information on which type of Product it is looking for. As with all factory pattern clients, the client here use the product object as abstract product without being aware of the concrete product implementation.

- Factory Method Pattern – GoF Factory pattern

The classic definition of Factory Method (also known as Virtual Constructor) pattern is "*Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.*"This is one of the widely used patterns in Java and .NET. One example in Java is *java.net.ContentHandlerFactory* interface. This interface defines a factory for content handlers. *URLConnection.getContent()* method uses this factory to get the appropriate content object.

This pattern is used, when the class wants its subclasses to specify the object it creates and when the class cannot anticipate the object class it needs to create. That means, the client doesn't know the type of the objects that needs to be created, still want to perform operations on them. Here the creator or factory relies on its *subclass*'s factory method to return instance of an appropriate concrete product object. The creator can execute some operations or sequence of operations on the object.This pattern decouples product sub class details from the client and new products can be added without affecting the existing code.

7. Solve (any four) :- [20 Marks]

(a) Difference between game and Business application.

If we want to start putting in some gameplay, the question "What is a game?" can serve as a good starting point.

- Cool Features

Cool features are just fine; in fact, they are a necessity. However, cool features do not, of themselves, make the game. You know those brainstorming sessions with your developers that end with everybody saying, "Wouldn't it be great if you could give a whole bunch of swordsmen an order and every swordsman would do something a tiny bit different?"

Even if all those great little ideas didn't take forever to implement, there's a point at which cramming in extra features just starts to damage the elegance of the gameplay. This tendency to add unnecessary features, commonly known as *gold plating*, is always the result of somebody, at some point, deciding those features would be cool. They may well be cool, and having them might help the game—just know when to draw the line.

- Fancy Graphics

Games need fancy graphics, just as a blockbuster movie needs special effects—but neither will save the product if the core creativity and quality is lacking. The fact is that in today's market, not having fancy graphics in your game is commercial suicide effectively because games are a chart-driven industry. All such industries are strongly affected by the reviewers—who tend, of course, to be hardcore aficionados with top-oft he- range machines and who therefore expect to see impressive technology on show.

The danger with fancy graphics (as with cool features) is that they can distract the development team's attention from putting any depth into the game. The movie industry is littered with examples of movies that cost $100 million and up but failed to spend anything of that on getting a good script. Cinemagoers can see through that, and there is growing evidence that gamers are starting to also. We would certainly never turn down any fancy graphics that the technology can provide, but the game has to be able to work without them.

- Puzzles

All games have puzzles. From determining where to build your castle extension to figuring out how to kill a wave of tricky aliens, games can universally be described as containing sets of linked puzzles.

You may or may not like puzzles. Heuristic puzzles can be diverting, but often that's just the problem: They divert attention from an interesting story or game. On the other hand, some people like them—although a "pure" puzzle game does not exist in the wild, as they are usually merged with another genre to boost the interest level. (A good example of this would be *Tetris*.) Either way, puzzles are not gameplay in themselves. Puzzles are specific problems. Game design is about creating a system that will spawn generic problems.

- Setting and Story

Again, who could bring themselves to drown this kitten? A good setting encourages *immersion*—what Coleridge called "the willing suspension of disbelief." And a good story draws the player in and impels him through the action. But, again, it will add up to nothing if the gameplay isn't there.

The most valuable skill a software analyst develops is to know when to stop. Knowing when a design is "a done deal" is what separates the real analysts from the wannabe newcomer. In your average business application, this mapping may not be so immediately obvious.
After all, trying to visualize a space containing tax returns, billing receipts, and bank accounts with transactions zipping through the ether does not come as easily the average human being as imagining a space full of spacecraft with lasers flashing between them. Of course, the token-to-object mapping may not always be so trivial; it can be deceptive.

Games are generally more intuitive and comprehensible than business applications. It's one of the things that makes them fun. The average player has to be able to relate to the world within the game; otherwise it is not fun. If the world within the game is easy for a wide range of people to relate to, then it is most probably expressed in terms of understandable concepts and entities.

The tokens in the game can be easily mapped to real (or imaginary) life equivalents, and the interactions between these tokens can be implicitly understood and predicted. Of course, this does not apply to all games or business applications. We know of enough games that are not fun—and enough business applications that are—to support this last point.

Business applications and games have two different agendas. A business application has no real goal: It is merely a tool to use to complete a task. You don't get three attempts at creating your spreadsheet before you have to go back to the beginning and start again (although with the early buggy releases of a certain famous spreadsheet application, it certainly seemed that way). A good game is meant to provide a challenge. A good business application is not.

(b) Software factory

The term *software factory* refers to a methodology of producing software with techniques similar to those used in a standard factory, such as one that manufactures cars. This doesn't mean, however, that all the software produced by this method will be identical, churned out endlessly, and devoid of imagination and flair. Enough companies are already too good at producing creatively bankrupt software.

As the computer games industry begins to mature, more sophisticated production methods, such as the software factory, are being implemented to develop new products. The software factory methodology centralizes and simplifies the production of specific common modules that can be used across several products.

It uses the advantages of mass production to make specific tasks both easier and more efficient, and it has the benefit of ensuring that a core set of tools and libraries are well maintained and supported over a series of products. Thus, subsequent products become easier to develop, as they will be supported by a more resourceful library and useful tool set.

The software factory has been proven to work time and time again on projects with common functionality . Keep in mind that this method may not be best suited to a particular product. (You or your co-workers may have a better methodology scrawled on the back of an envelope!) However, software factories are particularly well-suited for producing a series of products.

"But these are *games* we're writing! Each one is unique! There is no way we can rehash the code, change the graphics, and re-release the game!" This objection is perfectly valid, but how many times do you want to write screen and sound setup code, data file loaders, compression libraries, CD track playing code, finite-state machine code, menu code, or any other chunk of potentially reusable code from a long list of basic modules? Looking at it from a management perspective, how much time and money do you want to spend for specialists to write code that you already have? Not only does the code have to be written, it also has to be tested, integrated, and debugged.Some common tasks that should be placed in reusable modules are as follows:

- Data file loading
- Hardware setup

- Hardware configuration
- Software configuration
- Custom CODECs (compression and decompression)
- Encryption/decryption code
- Windowing and graphics features
- Basic AI components
- User input

The software factory concept eases tasks such as producing common libraries and tool sets, leaving the developers to concentrate on the code bits that *should* be written from scratch. The fastest work is the work that has already been completed and tested for full functionality.

(c) Code priority

At each level of project development, certain priorities drive the process. These priorities will almost certainly differ for certain areas of the project. For example, the priorities for the code to provide a menuing system is likely to have wildly different priorities than the code for a 3D engine would have. (If they don't, then there is something seriously wrong with the project.)

You need a project edict to establish coding priorities as part of the technical design, and the coder needs to follow these priorities. Otherwise, time is wasted reimplementing code that is not suitable for the purpose.

These priorities cannot be ignored, as they are the low-level forces that govern the direction of the project. At the highest level, project goals drive the project, but at the lowest levels—where all the action is—coding priorities drive code implementation. What do we need to know about priorities? What sort of considerations need to be taken into account?

- Speed
- Size
- Flexibility
- Portability
- Maintainability

- Speed

The need for speed is foremost in the minds of the majority of game developers. Games have to be fast. Each function needs to be pared down to the absolute minimum. Each needs to be optimized to the max. Well, that's what a lot of developers believe. And, if they are working on restricted platforms such as the GBA, with its relatively low-powered processor, then they will be right. Of course, some code (such as menuing code) wouldn't set such a high priority on speed. As an example, on the GBA, the next highest priority would be system architecture considerations and size.

On more-powerful platforms, including the PC and Consoles, the quest for speed is a bit of a wild goose chase. It simply isn't possible to optimize down to the instruction level in a reasonable period. The processors that these machines use (and the software they run) tend to be so complex that the execution speed of any particular block of code is pretty nondeterministic at the lowest levels of granularity. The only way to measure speed with any accuracy is to use stochastic method; that is, run the code to be tested several thousand times and take the average execution time.

You could do this sort of thing manually, but this doesn't provide you with much application-level information. The best solution is to use a profiling package such as Compuware's *TrueTime* or Intel's *VTune* on PC, which instrument the application and collect information as it is running. The resulting database can be reviewed in a hierarchical form, and you can see which functions are taking up the most time, as a percentage of either program execution time or the execution time for the calling routine.

The main use of this sort of program is not just to tell you how effective your optimizations are, but also to tell you where to optimize. It's a standard statistic that your program spends 90% of the time in just 10% of the code. It's just not worth focusing your attentions on areas of code that simply aren't executed that often. You'll get much better results by focusing on the 10% of commonly executed code than you will by working on the rest. Sure, you might have the fastest appearing menu in the west, but who's going to notice if the main game loop runs like a lethargic slug?

In regards to optimization, it should begin at the algorithm level and end at the op-code level only as a last resort. You'll get much better optimization results if you take a long, hard look at the

algorithms you're using than you will by concentrating on the op-code level, which is the highest granularity level in the system. Only when all the lower granularity level options have been exhausted should you shift up a granularity level.

- Size

Size is less of a priority on the more powerful platforms. This is because the amount of system resources available to the programmer are usually exhausted by other means long before code size becomes an issue. On a typical CD-ROM game for the PC, the program files themselves take up less than two or three megabytes of space. The data for the game—comprising the artwork, the music, and the game data—take up the major part of the space on the CD-ROM.

On the PC or Macintosh, unless you are coding for a specific purpose (such as components that are to be download over the Internet uncompressed), then code size is never a consideration, because the available level of local read/write storage is usually much more than a typical game requires. The only size limitation on the PC (and these are rapidly disappearing) is the amount of memory on the graphics cards, but, again, this is a matter of data rather than of code.

The bottleneck here is how much data can be shifted across the card bus and how much of that data will fit in the card's memory. processor cache considerations are for the most part redundant. It's no longer important to try and tune assembly loops so that they fit comfortably inside the processor's first-level cache. The last famous program that took these things into consideration was the original *Quake,* but this was written before the advent of the powerful 3D graphics acceleration cards.

However, on space-limited platforms that do not have large amounts of fast, on-board storage (such as the PlayStation 2 or the Gamecube), then code size becomes a much more important consideration.

- Flexibility

Code flexibility depends on where and how it is intended to be used. For example, linked lists are generally useful in many situations, so, if you wanted to implement a linked list, it would make

sense to use templates that allow it to be customized for use in other modules by instantiating a type-specific version of the code where required.

Where reuse is a consideration, flexibility of code is also important. The rule to follow when considering flexibility is that the lower the level—and the more generic the functionality—the more important it is to make flexibility a priority.

- Portability

For portability to be a consideration, the platforms being targeted must be broadly similar in capabilities. For example, it would not make sense to attempt to write C code that would be portable from the PC down to the GBA because of the huge difference in power between the two platforms. The concept of portability here is meaningless.

However, between platforms of arguably similar capabilities—such as the PlayStation 2, PC, and GameCube (and to some extent the PC)—the potential for cross-platform libraries is much greater. The similar power and capabilities of these platforms lend themselves to some degree of abstraction, meaning that common game code could be written in a portable style using a consistent interface to platform-specific code on each machine. The downside of this is that consumers can reject obvious "console ports."

- Maintainability

Maintainability—the readability and ease of modification—of code is usually one of the most important concerns, especially when the code is designed to be modified by more than one person. The software factory methodology places maintainability very high on the list of important priorities.

Maintainability is usually enforceable everywhere, but sometimes, when code is optimized for speed or size—or even to work around a bug—the maintainability can suffer slightly In general, it is best to mandate that code should be as maintainable as possible, except in cases in which the maintainability affects another higher priority. In this case augmenting
the documentation should make up this deficit.

(d) Graphic file formats

The primary web file formats are gif (pronounced "jiff"), jpeg ("jay-peg"), and, to a much lesser extent, png ("ping") files. All three common web graphic formats are so-called bitmap graphics, made up of a checkerboard grid of thousands of tiny colored square picture elements, or pixels. Bitmap files are the familiar types of files produced by cell phone and digital cameras, and are easily created, edited, resized, and optimized for web use with such widely available tools as Adobe's Photoshop or Elements, Corel's Paint Shop Pro and Painter, and other photo editing programs.

For efficient delivery over the Internet, virtually all web graphics are compressed to keep file sizes as small as possible. Most web sites use both gif and jpeg images. Choosing between these file types is largely a matter of assessing:

- The nature of the image (is the image a "photographic" collection of smooth tonal transitions or a diagrammatic image with hard edges and lines?)
- The effect of various kinds of file compression on image quality
- The efficiency of a compression technique in producing the smallest file size that looks good

➢ GIF Graphics

The CompuServe Information Service popularized the Graphic Interchange Format (gif) in the 1980s as an efficient means to transmit images across data networks. In the early 1990s the original designers of the World Wide Web adopted gif for its efficiency and widespread familiarity. Many images on the web are in gif format, and virtually all web browsers that support graphics can display gif files. gif files incorporate a "lossless" compression scheme to keep file sizes at a minimum without compromising quality. However, gif files are 8-bit graphics and thus can only accommodate 256 colors.

➢ PNG graphics

Portable Network Graphic (png) is an image format developed by a consortium of graphic software developers as a nonproprietary alternative to the gif image format. As mentioned above, CompuServe developed the gif format, and gif uses the proprietary lzw compression scheme, which was patented by Unisys Corporation, meaning that any graphics tool developer making software that saved

in gif format had to pay a royalty to Unisys and CompuServe. The patent has since expired, and software developers can use the gif format freely.

Png graphics were designed specifically for use on web pages, and they offer a range of attractive features, including a full range of color depths, support for sophisticated image transparency, better interlacing, and automatic corrections for display monitor gamma. Png images can also hold a short text description of the image's content, which allows internet search engines to search for images based on these embedded text descriptions.

Png supports full-color images and can be used for photographic images. However, because it uses lossless compression, the resulting file is much larger than with lossy jpeg compression. Like gif, png does best with line art, text, and logos—images that contain large areas of homogenous color with sharp transitions between colors. Images of this type saved in the png format look good and have a similar or even smaller file size than when saved as gifs. However, widespread adoption of the png format has been slow. This is due in part to inconsistent support in web browsers. In particular, internet explorer does not fully support all the features of png graphics. As a result, most images that would be suitable for png compression use the gif format instead, which has the benefit of full and consistent browser support.

- Jpeg graphics

The other graphic file format commonly used on the web to minimize graphics file sizes is the joint photographic experts group (jpeg) compression scheme. Unlike gif graphics, jpeg images are full-color images that dedicate at least 24 bits of memory to each pixel, resulting in images that can incorporate 16.8 million colors.

Jpeg images are used extensively among photographers, artists, graphic designers, medical imaging specialists, art historians, and other groups for whom image quality and color fidelity is important. A form of jpeg file called "progressive jpeg" gives jpeg graphics the same gradually built display seen in interlaced gifs. Like interlaced gifs, progressive jpeg images often take longer to load onto the page than standard jpegs, but they do offer the user a quicker preview.

Jpeg compression uses a sophisticated mathematical technique called a discrete cosine transformation to produce a sliding scale of graphics compression. You can choose the degree of compression you wish to apply to an image in jpeg format, but in doing so you also determine the

image's quality. The more you squeeze a picture with jpeg compression, the more you degrade its quality. Jpeg can achieve incredible compression ratios, squeezing graphics down to as much as one hundred times smaller than the original file. This is possible because the jpeg algorithm discards "unnecessary" data as it compresses the image, and it is thus called a "lossy" compression technique

(e) Game display technologies

GAMING ARCHITECTURE AND PROGRAMMING (GAP) DECEMBER 2012 INFORMATION TECHNOLOGY SEMESTER 8

Con.9134-12. **(REVISED COURSE)** **KR-4713**

(3 Hours) **[Total Marks: 100]**

N.B. : (1) Question No.1 is compulsory.
(2) Attempt any four questions from Q. Nos.2 to 7.
(3) Assume suitable data if necessary.

1. (a) Explain game development process. [10 Marks]

- Step One: Initial Planning

The Genesis Gaming Design and Marketing teams will meet with the client to determine the key concepts driving the development of a strategic game portfolio. This will address issues such as analysis of an existing portfolio, current demographics, additional player acquisition and retention, emerging trends and profiles. We will further discuss a range of themes, volatility levels, features, bonus games and any other aspect required for bespoke development of a strategic portfolio. It is also important that there is an overall marketing discussion to determine how the games can be used to further extend the client's brand.

- Step Two: Technical Review

Our Engineering Team will review documentation provided by the client to determine if there are any technical questions or matters that need to be addressed to best develop to a specific platform.This will ultimately save significant time in the integration process.

- Step Three: Initial Themes & Concept Art

We will submit themes, names, concept art and descriptions for consideration and approval or modification. We will then work with the client to determine portfolio development priorities. This allows the ability to set expectations as to game delivery. The client can then implement a schedule for marketing and release.

- Step Four: Features and Mathematics

Our Game Design and Mathematics departments will determine and create unique math models that best represent the client's objectives as stated through the analytical process and the client's objectives. Math will be verified and all required percentages will be provided.

- Step Five: Art and Creative Design

The Genesis Gaming Art Department will design and submit static art and design elements for approval. Subsequently, our Animation and Music Composition teams will finalize the game. This will require the client to provide appropriate documentation such as templates and other substantive game specifications for consistency.

Upon completion, our Demo team will provide a playable version of the game for additional review.

- Step Six: Integration

Upon approval, our Engineering Department will integrate the game to the appropriate provider platform. This requires the necessary documentation, such as an API, from the client along with server support, assuming it is not an "asset only" delivery. We place a significant emphasis on product assurance and quality control so the game will be very "client provider friendly."

- Step Seven: Delivery

All game assets, in the requested file formats, including all release documentation will be provided according to contractually specified delivery requirements. Genesis Gaming is also happy to provide any material that may be requested for the client's marketing campaign.Since success is our mutual objective, it is our goal to make certain that the client has all tools possible to promote the best exclusive content in the industry

(b) Explain Hard and Soft architecture. [10 Marks]

Architecture Design

When we begin to consider the actual architecture of the game (refer figure 2.1), we need also to consider how to construct it around the planning needs of the schedule. "Milestones and Deadlines". Each milestone will specify the technical requirements to complete a particular tier. As we have been

discussing, the architecture is specified in three main stages, each of which expands into a number of tiers.

The three stages are:

- Prototyping :

The prototyping stage allows us to have a dress rehearsal of the full architecture, allowing us to tackle any tricky points and difficulties that we might encounter. Of course, this doesn't mean that we are going to be able to cover all of these difficulties, but we can tackle at least the more obvious ones, and, of course, we will be able to explore gameplay issues sooner than we would be able to otherwise.

- Hard-architecture design :

The hard-architecture design stage involves the laying of the game framework. However, the point of using a component- based design is to produce a set of generic components that can be used across projects. Hard-architecture components would need to be upgraded and augmented in order to keep up with emerging technology, but, with a sensible set of interfaces, the disruption caused by this continual upgrade (not replacement) process would be minimal. Only after a few projects have been undertaken will we see any benefits from reusing in-house components.

Once a few projects have been completed, we will also be able to use our own components.

- Soft-architecture design :

"The Software Factory," we mentioned that the software factory architecture caused an apparent drop in productivity for the first project developed using it, by applying more structured development methods that reduce the amount of time spent on backtracking and unnecessary rework. This is pretty much unique for any project, and this is as it should be, for within the soft architecture is the unique spark that makes your game stand out from the rest. By using the soft-architecture system, we are taking advantage of all the groundwork that has already been done for us. The soft architecture defines the game-specific functionality and data required, such as in game graphics, music, and other data. In fact, the soft architecture is essentially data driven.

2. (a) What are tokens? Explain tokenization with an example. [10 Marks]

In general, a token is an object that represents something else, such as another object (either physical or virtual), or an abstract concept as, for example, a gift is sometimes referred to as a token of the. We can also consider these tokens to be arranged in a form of hierarchical structure.

The playing area, or game world, in itself is at the top of the hierarchy. From then on in, it is an essentially flat hierarchy.

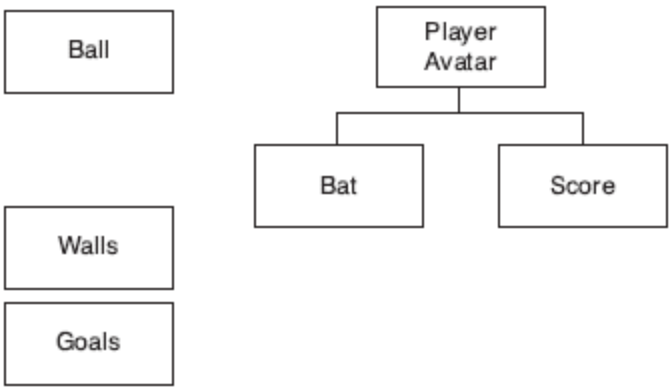


Figure :The *Pong* token hierarchy.

The game world token contains all the other tokens. Obviously every token has to operate within the game world in order to form a part of the game. The player avatar token is the representation of the player within the game world. It is effectively a channel for the user interface between the player and the game.

The player avatar for *Pong* is very simple; it is merely a bat and a score. These are how the player is represented in *Pong*. The other tokens—those manipulated by the computer—are the ball, the walls, and the goal zones.

Now it's time for a little sleight of hand of the sort that is possible with only the written word. Reread the two paragraphs, and for every instance of the word "token," read it as "object."

So, if we were just talking about objects all along, why didn't we just use the word "object" to start with?

The main reason—and why we particularly like the use of the word token and why we will continue using it from here on in—is that these conceptual tokens may not have a one-to-one mapping with the programming language objects (for example, in C++) that are defined by the programmers. What we are trying to do is to break down the game design into conceptual objects that will eventually

be translated into programming language objects. This tokenization process is an intermediary stage in the production of a decent architecture. In order to describe this without causing confusion, we need to use different terminology for each type of object.

The tokenization of a game design such as *Pong* is fairly trivial, and there's really only one way to do it. In spite of this, it makes an excellent example to try to demonstrate the thought processes behind tokenization. Not all games will be so trivial, and, for some more complex games, there may be many ways, all of which are equally valid. So now we have a set of tokens. On their own, they are not very exciting as they do not interact with one another. But as we know, in *Pong* there are all sorts of interactions going on. Well, one anyway: collisions.

We can now define an event—the collision event. Let's say that a collision event is generated when two tokens collide. The net result of this event is that each token receives a message telling it that a collision has occurred, and the type of object it has collided with.

The token interaction matrix is a very important construct. It is a chart of all the interactions that take place in the game. Note that for very large games we would not use a token-token matrix directly. Instead we would introduce an extra layer of abstraction by using token-property and/or property-property.

Okay, so let's look at the *Pong* token interaction matrix. The matrix is arranged in a triangular format, with each token listed along the side and the bottom. An unusual feature of *Pong* is that tokens do not come into contact with other tokens of the same type. This immediately means that the token-token interactions for bat-bat, ball-ball, wall-wall, goal-goal, and score-score can be discounted. Due to the nature of the game, the following interactions can also be discounted: bat-goal, wall-goal, score-bat, score-ball, and score-wall.

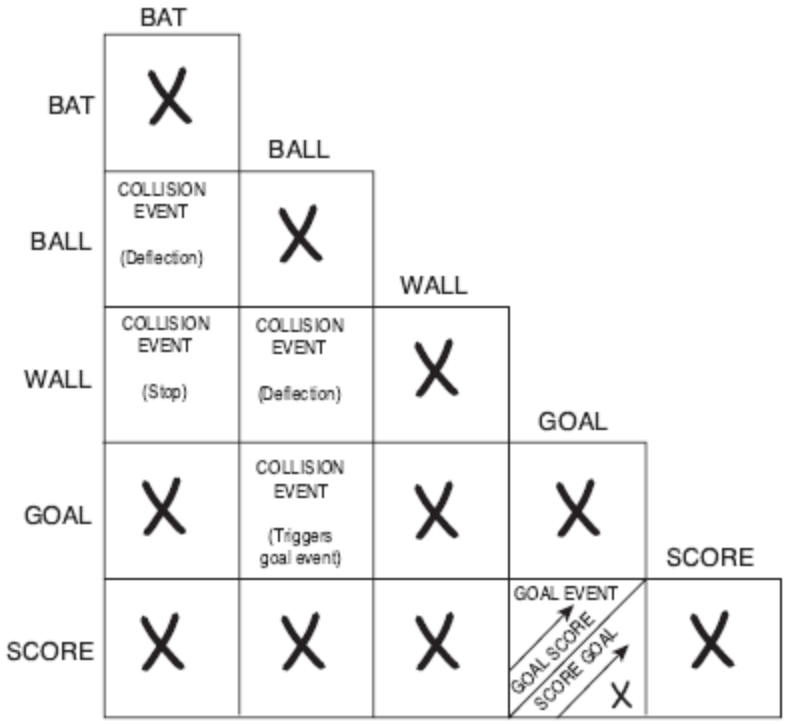


Figure : The *Pong* token interaction matrix.

(b) Explain source control system. [10 Marks]

A component of software configuration management, version control, also known as revision control or source control,[1]:2 is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number," "revision level," or simply "revision." For example, an initial set of files is "revision 1." When the first change is made, the resulting set is "revision 2," and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated, when the era of computing began. The numbering of book editions and of specification revisions are examples that date back to the print-only era. Today, the most capable (as well as complex) revision control systems are those used in software development, where a team of people may change the same files.

Version control systems (VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors and spreadsheets, e.g., Google Docs and Sheets[2] and in various content management systems, e.g., Wikipedia's Page history. Revision control allows for the ability to revert a document to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend against vandalism and spamming.

Software tools for revision control are essential for the organization of multi-developer projects.

In computer software engineering, revision control is any kind of practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code.

As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops).

Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently (for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used on many large software projects. While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of self-discipline on the part of developers, and often leads to mistakes. Consequently, systems to automate some or all of the revision control process have been developed.

Moreover, in software development, legal and business practice and other environments, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary interests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even indispensable in such situations.

Structure revision control manages changes to a set of data over time. These changes can be structured in various ways.

Often the data is thought of as a collection of many individual items, such as files or documents, and changes to individual files are tracked. This accords with intuitions about separate files, but causes problems when identity changes, such as during renaming, splitting, or merging of files. Accordingly, some systems, such as git, instead consider changes to the data as a whole, which is less intuitive for simple changes, but simplifies more complex changes.

When data that is under revision control is modified, after being retrieved by *checking out*, this is not in general immediately reflected in the revision control system (in the *repository*), but must instead be *checked in* or *committed.* A copy outside revision control is known as a "working copy". As a simple example, when editing a computer file, the data stored in memory by the editing program is the working copy, which is committed by saving.

Concretely, one may print out a document, edit it by hand, and only later manually input the changes into a computer and save it. For source code control, the working copy is instead a copy of all files in a particular revision, generally stored locally on the developer's computer in this case saving the file only changes the working copy, and checking into the repository is a separate step.

If multiple people are working on a single data set or document, they are implicitly creating branches of the data (in their working copies), and thus issues of merging arise, as discussed below. For simple collaborative document editing, this can be prevented by using file locking or simply avoiding working on the same document that someone else is working on.

Revision control systems are often centralized, with a single authoritative data store, the *repository*, and check-outs and check-ins done with reference to this central repository. Alternatively, in distributed revision control, no single repository is authoritative, and data can be checked out and checked into any repository. When checking into a different repository, this is interpreted as a merge or patch.

3. (a) State design patterns that are commonly used in game design.[10 Marks]

Design patterns are generalized solutions to generalized problems that occur with some modicum of frequency when you're creating software using the object oriented programming paradigm. Why Use Design Patterns? The most basic and condescending answer is: because these solutions have

existed for a relatively long time and many experts have used them, they're likely better than any solution you could come up with on your own.

And even if you did come up with a solution on your own, it's likely already a design pattern in some way

.Knowledge of contextually pertinent design patterns helps you to make good architecture and design decisions.

Common Game Programming Patterns

- Singleton - You create objects that ensure that only a single instance of which can exist at a time. In my game Total Toads, this is the design pattern used because it was easiest to fit with cocos2d's design. For example, in cocos2d, there's a Singleton CCDirector, CCSpriteFrameCache, etc. It seems this is an often-used panacea in game programming.

  Though the general consensus seems to be that it isn't a panacea so much as it is a cancer because it actually masks poorly designed architecture. You should probably avoid using this design pattern if you can because there's likely a better way to design the architecture of your game. This can avoid the problem of having multiple instances of objects of which there should only be one, like a "Player" object in a single player game.

- Factory - You create an object whose purpose it is to create other objects. For example, you can have a factory class called "GameObjectFactory" with static (possibly parameterized) methods to create other game objects like a "Player", "Enemy", "Gun", or "Bullet". The latter classes might have complex constructions that make obtaining a instance of that specific class difficult.

  The factory can take care of the object's complex configuration (adding to an object pool, adding to physics engine, etc) and simply return a reference to the created object. This pattern helps you avoid the problem of complex object instantiation by keeping these complex configurations in a single place, rather than scattered around in your code.

- Observer - The object in question maintains a list of other objects that are interested in its state and notifies these listening objects of a change in its state. In Total Toads, we have 3 Frog

objects and 3 FrogAnimation objects that can control the animation of the frogs based on their state. In this case, the FrogAnimation objects are observers of the Frog objects.

Every time the state variable for a Frog changes, it notifies the associated FrogAnimation object to notice the new state and take an action if necessary (like animating the frog). This pattern helps you avoid the problem of event notification in your game. Since games are user-interaction driven, objects can change state at almost any time. When an object changes state oftentimes that object needs to be animated or have other objects change their state with respect to the new state.

- State - You have an abstract (empty implementation) class that has subclasses to define the current state. An example of this might be a first person shooter that has a "Player" class who has several possible states like "PlayerInCombat", "PlayerOutOfCombat", and "PlayerInMenu".

When the state is changed, the player shall be represented as an instance of the appropriate state class. When the player starts to be shot at, the player object "switches" to an instance of the PlayerInCombat class to take advantage of that class's implementation of the mouse's left button click which makes the player able to shoot their gun. Similarly for the "PlayerOutOfCombat" and "PlayerInMenu" classes, where the mouse might allow usage of items or the clicking of menu options, respectively.

This pattern helps you avoid monolithic methods that perform differently based on the object's state by having a bunch of switch or if/elses in your code. Instead, the object just changes and runs its appropriate code. This also simplifies your code and allows you to more easily find problems in your objects behavior since the state is right in the object's class name.

(b) What are the research goals and explain blue sky research. [10 Marks]

We would all love to spend all day in front of a computer munching pizza and guzzling coffee, freely engaging our creativity and researching whatever we like. Some companies do allow this and have active (if somewhat variable) research departments. That’s a good thing. Research is essential to the survival of a company. It’s *blue-sky* research that you have to be careful about. By definition, any

technology present in games currently available in the shops is about 18 months behind what is currently being worked on by the best design houses.

In other words, even with the best will in the world, it is statistically unlikely that anything remotely useful will be achieved, except a rather tragic comedy of errors. When a game project depends on the outcome of research that has not been completed, that project is in great danger. Putting anything in the critical path for which the outcome cannot be predicted is sheer idiocy; but for some reason—whether greed, stupidity, or just plain ignorance—development teams seem to do this on a regular basis.

We're not advocating that research should be abolished; that would be a draconian measure and would lead to further stagnation of the industry. What we *are* saying is that there is a time for research and a time for development, and that the two should never overlap. Any research that is instigated should be directed research. It should have an aim.

An example would be the design and development of a library useful in fuzzy-logic calculations. For sure, there would be a fair amount of research involved, but this would be research into ways of optimizing and improving on known techniques. The blue-sky alternative would be to look for a completely new method of doing things. Fortunately, there is very little you can imagine doing with a computer that hasn't been already done by someone somewhere.

If you're very lucky, then they have published their results. Building on the work of others, although less glamorous, is a surer way of getting good results. Besides, if you wanted glamour then why did you become a developer? When your team is researching, set a strict time limit and stick to it ruthlessly. Run with what you physically have at the end of the research period, not with what you are promised at the beginning.

The time allocated for research should also factor in the time necessary to bring the research project to a successful conclusion and with a stable and well-documented component. As the research progresses, more and more unknowns will become quantifiable. As soon as the fundamental technique being researched is working, a mini-schedule can be drawn up to allow for this tidying-up procedure. If the game design depends on this research, then this is the single most critical point of the whole development.

The project will succeed or fail—right here, right now—dependent on the success of this research project. Obviously, this is not a good idea: Gambling the whole project on blue-sky research is a game that only the very foolhardy or the extremely desperate would play. Remember, any safety net is

better than no safety net. If no safety net is possible, then "make-or-break" research should be avoided. It's too much of a risk, no matter how cool the developer doing the research is.

Under most circumstances, releasing the product is better than canning it. Only the most cash-rich companies will be able to afford internal research such as that used during the development processes of *Die by the Sword* and *Outcast.* Not so many companies are able to afford the protracted development times and the risks that are associated with research of this nature. If your company doesn't have the sort of cash available to finance a research department, you still have a few options open.

The first, and most undesirable, is to avoid projects that require R&D. This is only really acceptable for the "ticking-over-and-we-know-we'll-only-sell-a-couple-of hundred- thousand-or-so" type of company. This is a viable option, but it isn't going to go anywhere fast.

For the less-affluent companies, another option is to form links with academia. Forming a loose relationship with universities and their computing departments can provide some good technology at some very good prices. A company that we have worked for in Belgium used their local university as a source of cutting-edge cryptography algorithms and regularly tested their latest algorithm designs by letting the university researchers loose on them. The theory was that any implementations of algorithms that these university cryptography experts couldn't crack were pretty much guaranteed to be secure.

If you use the "external research" system, then this also simplifies some other company decisions. Form a liaison with academia: A good relationship means that they get to publish and you get to use their new ideas. Yet another option is to be part of a large conglomerate, either using the Hollywood studio model or the loose alliance model typified by Lionhead's satellite. The latter may not be an optimal system.

To optimize it you need to ensure that the separate development teams are aware of and able to share each other's technology. Also you would ideally organize collateral R&D so that two companies don't waste time on the same tasks, all of which is best achieved by appointing a resource investigation unit to interface between all the projects in development. A loose development alliance can work, therefore, but it needs structure.

4. (a) Explain various platforms on which game can be deployed on. What are the advantage and disadvantages of each of these platform? [10 Marks]

Finding the right game engine can be the key to successfully building and deploying a game that becomes both popular and lucrative. But there are so many game engines out there vying for your attention. Clearly, some guidelines on the subject would be useful.

Ten years ago, it was okay to release your game on one platform at a time. Today, it's more typical for a game to be released rapidly on multiple platforms. To that end, a cross-platform game engine offers some real advantages, and the options there are quite diverse and plentiful. Having recently released my own game template based on Cocos2D JS, I thought it would be interesting to compare some of the major game engines and see how they stack up against each other.

To prepare for this post, I wrote a complete Breakout clone in four of today's top cross-platform game engines: Unity, Corona, Cocos2D JS and Appcelerator Titanium, and also using my game template, RapidGame Pro. The code can be found at the end of each section, so you can see for yourself. My observations on how they all compare should help you make a choice that may save you and your team weeks or months.

- Unity : Unity is, in short, a closed-source, cross-platform game development application. You create your game by manipulating objects in 3D and attaching various components to them. Even 2D games must be manipulated in 3D. Scripts are written in C# (recommended), Boo or Unityscript (mistakenly called JavaScript) and attached to 3D objects as components.

  Launching Unity for the first time, you may feel like the pilot of a 747 jet plane. There is much to learn before even the first switch can be flipped. First of all, there's camera and lights. When trying to add a simple cube to the scene, it can get lost behind the camera or perhaps be invisible because there's no light. In short, there is a learning curve.

  Prepare to spend approximately 8-12 hours getting familiar enough to develop your own game.Unity was first released in 2005 and the interface hasn't changed much since. To be frank, it feels like many of the repetitive tasks in day-to-day Unity game development are busy work. Adding audio sources, updating prefabs and importing assets are all examples of tasks that shouldn't have to be done, or shouldn't take so long. It would be nice if Unity had a modern makeover.

  That said, once you've created a game with Unity, deployment is a cinch. With a couple of clicks, you can export your game to mobile, desktop and/or web (web currently requires the

Unity player app to be installed). If you have the right license, you can even deploy to gaming consoles like Xbox, Playstation and Wii.

- Corona : Corona is a closed-source 2D game simulator and cloud-build application. Game code is written in Lua scripts and played back in the Corona simulator. Like Mystique from X-Men, the simulator can take on many skins, resolutions and ratios. When you're ready to deploy, it builds your game in the cloud and delivers you an iOS or Android game client.

Ah, the sweet joy of developing games with Corona. Everything about the language is easy. Adding a physics body, for example, takes only one line of code. After a mere 2-4 hours of getting familiar with the platform you'll be ready to develop games. And once you start it's difficult to stop. The simulator is responsive, quick and polite about using your computer's resources. With the simulator and your choice of code editor open side by side, you can save the Lua file and the simulator instantaneously reloads the game. It's simply delightful to develop a game with such rapidity.

One shortcoming of Corona is its limited deployment options. Only mobile platforms like iOS, Android, Kindle and Nook are supported. Windows Phone is coming soon. Cloud-Imagine a day full of testing your game on the device, tweaking one little thing and waiting a few minutes to be able to see if it worked.

Like Unity, Corona is closed-source and proprietary. There's no way to make a modification or fix a bug in the engine, and you cannot learn from its code.

- Cocos2D JS : Cocos2D JS is a cross-platform, open-source, free game development SDK. It is the newest — and perhaps sexiest — member of the Cocos2D family. Essentially it's a combination of two popular open-source projects: Cocos2D X for mobile / desktop and Cocos2D HTML5 for web. While it is currently 2D / 2.5D, there are plans to add 3D support.

You write game code entirely in JavaScript. On native platforms like mobile and desktop, your game's JavaScript is bound to native C++ objects, granting you maximum speed without having to write any native code. Web platforms run pure JavaScript and render using Canvas or WebGL, so no player applications need to be installed.

The easiest way to get started with Cocos2D JS game development is using the HTML5 platform. Open up a browser window and your favorite text or code editor, save your

JavaScript, refresh the browser and voila. It's a rapid way to develop. When you're ready to test and deploy to native platforms, you'll need Xcode, Visual Studio and/or Eclipse.

Cocos2D JS games can currently be deployed to iOS, Android, Blackberry, Windows Phone, Mac, Windows, Linux and HTML. With such wide deployment options, it's easy to see why many game developers are choosing Cocos2D.

- Appcelerator Titanium : Titanium is a cross-platform, open-source app development kit and Eclipse-based IDE. Apps are written in JavaScript and run natively, not just in a WebView. With Titanium Studio it's possible to develop, test and deploy to mobile and web platforms.

For 2D game development, there's the Platino Game Engine, an open-source — but not free — SDK that can be added to your Titanium stack. Getting acquainted with Titanium (more specifically Platino) is not as easy as it could be. The documentation has holes. For example, the crucial .center sprite property is left undocumented. Moreover, the physics engine is cumbersome and archaic. You have to synchronize all physics bodies and sprites manually using a very non- JavaScript, C-like API.

On the bright side, one nice thing about Titanium development is that the SDK is prebuilt. You can run your game on a simulator or device with very short build times.

- RapidGame Pro : RapidGame Pro, an open-core game (dual MIT licensed) template based on Cocos2D JS, to make game development using open source more rapid. It achieves this in a few ways:

By providing a project creator tool and game templates that make starting a game with scenes, sprites, sound, physics, a server, monetization, social, etc. a breeze.By prebuilding native libraries.

By providing and incorporating plugins for IAP, displaying ads, social networking, analytics, asynchronous multiplayer and virtual economies that work on *all* platforms.By including example code to a complete game based on multiple currencies.

Some of RapidGame Pro's plugins had to be developed from scratch for multiple platforms. For example, the Facebook plugin — including both social networking and IAP via Facebook Payments — is written separately in C++ for iOS, C++ and Java for Android, and

JavaScript for HTML5. All of these implementations are accessible from your game using write code for and test.

Likewise, the following code will display a full-screen video advertisement. Behind the scenes, this single JavaScript API call runs native C++, Java or JavaScript, depending on the platform.

RapidGame Pro helps perform day-to-day development tasks faster. The Cocos2D X libraries and plugins are prebuilt, so when you run your game in the simulator or on the device it will launch almost instantaneously.

Developing your own game template with social networking, monetization and other plugins for multiple platforms — for even just *one* platform — can take months to get right.RapidGame Pro lets you start with all the little things a pro- grade game needs already done.

For a game developer, choosing the right cross-platform game engine can be the single most important decision they make. I hope my insights help you to make that choice.

(b) What are the three stages of running a game? Explain in details. [10 Marks]

5. (a) Describe the 3D graphic pipeline in detail. Explain the various inputs to this pipeline and the operations performed on it by graphics pipeline. [10 Marks]

In 3D computer graphics, the graphics pipeline or rendering pipeline refers to the sequence of steps used to create a 2D raster representation of a 3D scene. Plainly speaking, once a 3D model has been created, for instance in a video game or any other 3D computer animation, the graphics pipeline is the process of turning that 3D model into what the computer displays.

In the early history of 3D computer graphics, fixed purpose hardware was used to speed up the steps of the pipeline through a fixed-function pipeline. Later, the hardware evolved, becoming more general purpose, allowing greater flexibility in graphics rendering as well as more generalized hardware, and allowing the same generalized hardware to perform not only different steps of the pipeline, like in fixed purpose hardware, but even in limited forms of general purpose computing.

As the hardware evolved, so did the graphics pipelines, the OpenGL, and DirectX pipelines, but the general concept of the pipeline remains the same. The 3D pipeline usually refers to the most common form of computer 3D rendering, 3D polygon rendering, distinct from ray tracing, and

raycasting. In particular, 3D polygon rendering is similar to raycasting. In raycasting, a ray originates at the point where the camera resides, if that ray hits a surface, then the color and lighting of the point on the surface where the ray hit is calculated.

In 3D polygon rendering the reverse happens, the area that is in view of the camera is calculated, and then rays are created from every part of every surface in view of the camera and traced back to the camera Computers began undergoing a significant change in recent years with the introduction of a separate video card and the rise of hardware accelerated graphics. This has led to the need for a programmable graphics pipeline which can be manipulated by shaders

Since the introduction of the programmable graphics pipeline most fixed-function pipeline implementations have become obsolete, such as OpenGL's immediate mode, or Direct3D's built in hardware Transform, clipping, and lightingThe Direct3D 11 programmable pipeline is designed for generating graphics for realtime gaming applications. This section describes the Direct3D 11 programmable pipeline. The following diagram shows the data flow from input to output through each of the programmable stages.

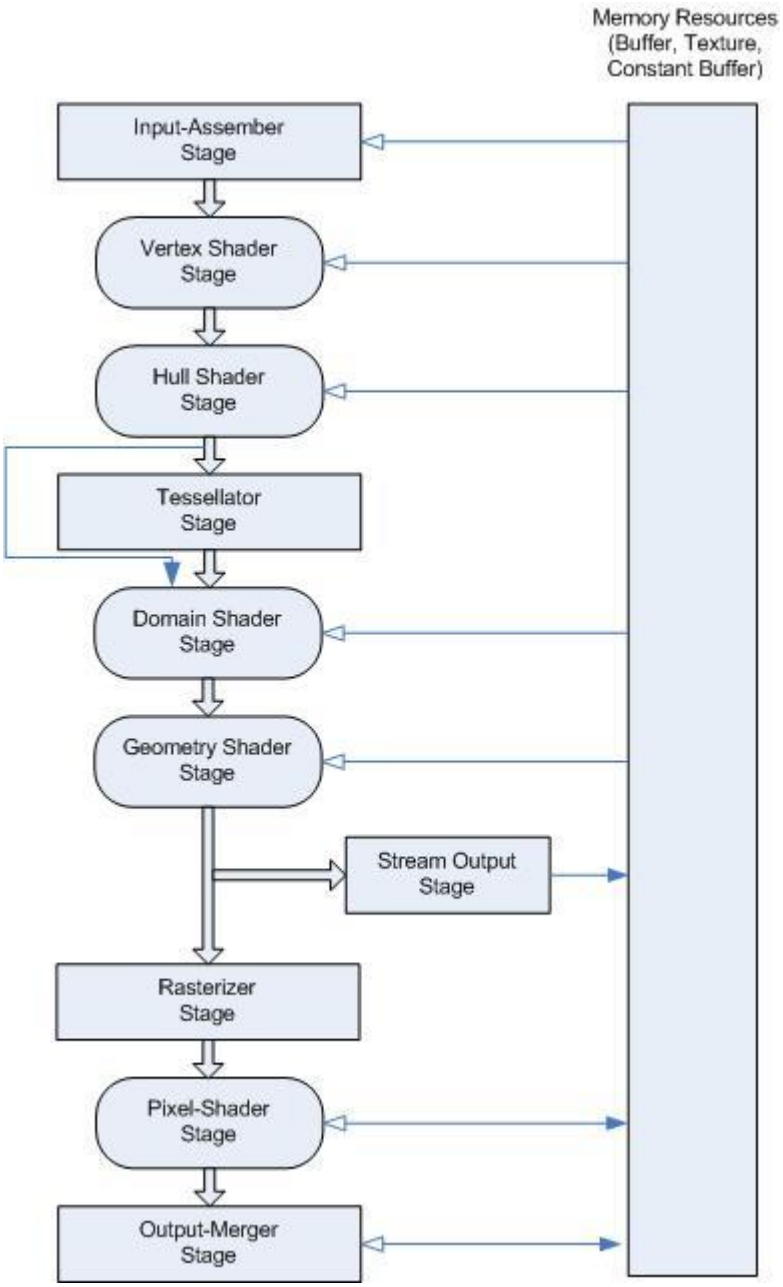


Figure : Graphics pipeline

The graphics pipeline for Microsoft Direct3D 11 supports the same stages as the Direct3D 10 graphics pipeline, with additional stages to support advanced features.

You can use the Direct3D 11API to configure all of the stages. Stages that feature common shader cores (the rounded rectangular blocks) are programmable by using the HLSL programming language. As you will see, this makes the pipeline extremely flexible and adaptable. The following list specifies the purpose of each of the stages.

- Input-Assembler Stage - The input-assembler stage supplies data (triangles, lines and points) to the pipeline.

- Vertex-Shader Stage - The vertex-shader stage processes vertices, typically performing operations such as transformations, skinning, and lighting. A vertex shader always takes a single input vertex and produces a single output vertex.

- Geometry-Shader Stage - The geometry-shader stage processes entire primitives. Its input is a full primitive (which is three vertices for a triangle, two vertices for a line, or a single vertex for a point). In addition, each primitive can also include the vertex data for any edge-adjacent primitives.

  This could include at most an additional three vertices for a triangle or an additional two vertices for a line. The geometry shader also supports limited geometry amplification and de-amplification. Given an input primitive, the geometry shader can discard the primitive, or emit one or more new primitives.

- Stream-Output Stage - The stream-output stage streams primitive data from the pipeline to memory on its way to the rasterizer. Data can be streamed out and/or passed into the rasterizer. Data streamed out to memory can be recirculated back into the pipeline as input data or read-back from the CPU.

- Rasterizer Stage - The rasterizer clips primitives, prepares primitives for the pixel shader, and determines how to invoke pixel shaders.

- Pixel-Shader Stage - The pixel-shader stage receives interpolated data for a primitive and generates per-pixel data such as color.

- Output-Merger Stage - The output-merger stage combines various types of output data (pixel shader values, depth and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.

- Hull-shader, tessellator, and domain-shader stages, which comprise the tessellation stages - The tessellation stages convert higher-order surfaces to triangles for rendering within the Direct3D 11 pipeline.

The Direct3D 11 programmable pipeline is also designed for providing high-speed computing tasks. A compute shader expands Direct3D 11 beyond graphics to support general purpose GPU computing.

(b) What are the core groups in software factory and their interactions? [10 Marks]

A glue language is a programming language (usually an interpreted scripting language) that is designed or suited for writing glue code – code to connect software components. They are especially useful for writing and maintaining:

- Custom commands for a command shell
- Smaller programs than those that are better implemented in a compiled language

"Wrapper" programs for executables, like a batch file that moves or manipulates files and does other things with the operating system before or after running an application like a word processor, spreadsheet, data base, assembler, compiler, etc.

- Scripts that may change
- Rapid prototypes of a solution eventually implemented in another, usually compiled, language.

- GUI scripting

With the advent of graphical user interfaces, a specialized kind of scripting language emerged for controlling a computer. These languages interact with the same graphic windows, menus, buttons, and so on that a human user would. They do this by simulating the actions of a user. These languages are typically used to automate user actions. Such languages are also called "macros" when control is through simulated key presses or mouse clicks, as well as tapping or pressing on a touch-activated screen.

These languages could in principle be used to control any GUI application; but, in practice their use is limited because their use needs support from the application and from the

operating system. There are a few exceptions to this limitation. Some GUI scripting languages are based on recognizing graphical objects from their display screen pixels. These GUI scripting languages do not depend on support from the operating system or application.

- Application-specific languages

Many large application programs include an idiomatic scripting language tailored to the needs of the application user. Likewise, many computer game systems use a custom scripting language to express the programmed actions of non-player characters and the game environment. Languages of this sort are designed for a single application; and, while they may superficially resemble a specific general-purpose language (e.g. QuakeC, modeled after C), they have custom features that distinguish them. Emacs Lisp, while a fully formed and capable dialect of Lisp, contains many special features that make it most useful for extending the editing functions of Emacs. An application-specific scripting language can be viewed as a domain-specific programming language specialized to a single application.

- Extension/embeddable language

A number of languages have been designed for the purpose of replacing application-specific scripting languages by being embeddable in application programs. The application programmer (working in C or another systems language) includes "hooks" where the scripting language can control the application. These languages may be technically equivalent to an application-specific extension language but when an application embeds a "common" language, the user gets the advantage of being able to transfer skills from application to application. A more generic alternative is simply to provide a library (often a C library) that a general-purpose language can use to control the application, without modifying the language for the specific domain.

JavaScript began as and primarily still is a language for scripting inside web browsers; however, the standardization of the language as ECMAScript has made it popular as a general purpose embeddable language. In particular, the Mozilla implementation SpiderMonkey is embedded in several environments such as the Yahoo! Widget Engine. Other applications

embedding ECMAScript implementations include the Adobe products Adobe Flash (Action Script) and Adobe Acrobat (for scripting PDF files).

6. (a) What are Direct-X and open GL? Microsoft did not want to continue supporting DOS. It saw the future as a 32-bit

Graphical operating system that was fast enough to support games. In what could charitably be considered to be practice run, they released WinG, a form of the Game SDK (and a precursor to DirectX), designed to allow faster access to hardware resources on Current Development Methods

Windows-based machines, we can remember only one game, *SimTower*, that was commercially released using this (although we're sure that there were probably more), and the game suffered from appallingly slow graphic updates.

WinG died a quiet death, and, meanwhile, Windows 95 was released with a fair amount of pomp in the closing months of 1995, as we're sure some of you will remember, Windows 95 sounded the official Microsoft death knell for 16-bit applications; DOS was dead.

Except, that is, if you were writing a game. The games industry practically ignored this new operating system, except to check it for compatibility with their Dos4gw executables. It was not considered then as a serious target for game development. Microsoft, on the other hand, did not want to continue supporting DOS, seeing the games industry as the last bastion of the DOS programmer.

If Microsoft could persuade the game developers to begin producing games for Windows 95, it will have achieved its aim of phasing out DOS, and shown the world that Windows 95 was the operating systemof the future. To this end, Microsoft developed DirectX, a library to allow the game developer direct access to the hardware if required, and to take advantage of any acceleration provided in the hardware.

The ideology behind DirectX was to provide a standard interface to the hardware that was easier to access than had been traditionally possible using DOS. Initially DirectX was touted by Microsoft as being potentially faster than DOS for graphics, because it would automatically take advantage of any hardware acceleration that was available.

DirectX spawned mild interest (and derision), but it was not until the release of DirectX 2 that people took notice. (Maybe it was the lurid yellow CD with a black radiation warning symbol that

caught their eye.) With DirectX 2, the API had just about become usable, and was taking advantage of the newly developed Microsoft Component Object Model technology (COM). COM is an object-oriented technique that allows objects to be easily versioned and (at least in theory) accessible from languages other than that in which the object was originally written.

During this period, Microsoft had not been lazy in updating its C/C++ compilers. It was painfully aware that Visual C++ 2.x (which compiled for 32-bit applications) was showing its age, and consequently an update was released in the form of Visual C++ 4.

This was a bit of a shock to the industry: a whole "generation" of game programmer had grown up with the concept of "Microsoft = slow, Watcom = fast," and they were surprised to discover that the Microsoft compiler produced faster, tighter code than the Watcom compiler. The result of this was the gradual transferal of game development from targeting DOS to targeting Windows 95. The first game released as a Windows 95 native executable, taking advantage of DirectX functionality, was a version of the classic Activision game, *Pitfall.* The initial release of DirectX was not fully compatible with the Watcom compiler, forcing anyone who wanted to do serious game development to switch to the Microsoft compiler. We'll leave it for the reader to decide whether this was by accident or design.

With the increasing momentum of Windows 95 game releases, Apple, manufacturers of the Macintosh line of computers couldn't sit back and let its (already tenuous) position be eroded further. More and more games were being released for Windows *95,* which further strengthened Windows' position as the operating system for the masses.

To fight this onslaught, Apple released the Game Sprockets, a set of components analogous to (but not compatible with) DirectX for the Macintosh operating system. How successful this was still remains to be seen, but, to date, only a handful of companies (Blizzard and Bungie being two examples that spring to mind) are releasing games for both platforms. We'll plead the fifth on whether this is due to the difficulty of writing cross-platform code, or whether it is due to lack of sales on the Macintosh.

Recently, however, Apple has reversed its fortune with the relative success of the "designer" computer, the iMac, but whether this will impact the games market in any significant fashion remains to be seen. Since the first imprint of this book, the Game Sprockets have come and gone with little attention, and Apple is now focusing on their OS X platform with integrated OpenGL and multimedia support. Although this hasn't pushed them to the forefront of gaming, a steady stream of titles are making their way to the Mac, including such notables as *Warcraft III* and *Neverwinter Nights*.

This just about brings us up to the present day, at least for home computer development DirectX has matured through nine versions, and there is no serious PC game development done without it, except in the case of certain OpenGL based exceptions— which are becoming more plentiful nowadays. The Microsoft compiler has been substantially enhanced and now is considered to produce the fastest and most compact code and have the friendliest user interface.

Advantages such as network and multipoint it or debugging and "edit and continue" (which lets the developer modify code on the fly without fully recompiling the program) have served to make the life of the game developer easier—at least in some ways. The growing power and diversity of computer systems have also made the developer's life more difficult.

The variety of hardware, and the subsequent complexity of programming the API—compounded by the amount of new information to absorb—is definitely more than before. It's like the development of scientific knowledge: In the eighteenth century, it was possible for one person to know all the areas of science in depth. Now, even after years of study, the best that could be hoped for would be to specialize in a tiny area.

(b) Explain different types of game genre with an example. [10 Marks]

A video game genre is a specific category of games related by a common gameplay characteristic. Genres are not usually defined by the actual content of the game or its medium of play, but by its common challenge.[1]

Genres may encompass a wide variety of games, leading to even more specific classifications called *subgenres*. For example, an action game can be classified into many subgenres such as platform games and fighting games. Some games, most notably browser and mobile games, are commonly classified into multiple genres.

The following is a list of all commonly-defined video game genres, with short descriptions for individual genres

Action games emphasize physical challenges that require eye-hand coordination and motor skill to overcome. They center around the player, who is in control of most of the action. Most of the earliest video games were considered action games; today, it is still a vast genre covering all games that involve physical challenges.

Action games are classified many subgenres. Platform games and fighting games are among the best-known subgenres, while shooter games became and continue to be one of the dominant genres in video gaming since the 1990s. Action games usually involve elements of twitch gameplay.

- Platform games

  Platform games are set in an environment with platforms, hence the name *platform game*.Platform games (or *platformers*) are set in a vertical or three-dimensional (3D) environment. Players guide a character through obstacles, jumping on platforms and battling enemies in order to advance. They often involve unrealistic physics and special movement abilities.

  Donkey Kong was one of the earliest and best-known platformers; the American gaming press classified it using the term *climbing game* at the time *Super Mario Bros.* was one of the best-selling games of all time; more than 40 million copies were sold (excluding Game Boy Advance and Virtual Console sales). *Jumping Flash!* introduced 3D graphics to the genre, being the first console platformer to incorporate 3D graphics.

- Action-adventure

  Action-adventure games combine elements of their two component genres, typically featuring long-term obstacles that must be overcome using a tool or item as leverage (which is collected earlier), as well as many smaller obstacles almost constantly in the way, that require elements of action games to overcome. Action-adventure games tend to focus on exploration and usually involve item gathering, simple puzzle solving, and combat. "Action-adventure" has become a label which is sometimes attached to games which do not fit neatly into another well known genre.

  The first action-adventure game was the Atari 2600 game *Adventure* (1979). It was directly inspired by the original text adventure, *Colossal Cave Adventure*. In the process of adapting a text game to a console with only a joystick for control, designer Warren Robinett created a new genre. Another typical Action-Adventure game is "The Legend of Zelda" by Nintendo, which involves puzzle solving, sword fighting, and item collecting. Because of their prevalence on video game consoles and the absence of typical adventure games, action-adventure games are often called "adventure games" by modern gamers.

- Stealth game

Stealth games are a somewhat recent subgenre, sometimes referred to as "sneakers" or "creepers" to contrast with the action-oriented "shooter" subgenre. These games tend to emphasize subterfuge and precision strikes over the more overt mayhem of shooters, for example, the Sly Cooper series.

- Adventure

Adventure games were some of the earliest games created, beginning with the text adventure *Colossal Cave Adventure* in the 1970s. That game was originally titled simply "Adventure," and is the namesake of the genre. Over time, graphics have been introduced to the genre and the interface has evolved.

Unlike adventure films, adventure games are not defined by story or content. Rather, adventure describes a manner of gameplay without reflex challenges or action. They normally require the player to solve various puzzles by interacting with people or the environment, most often in a non-confrontational way. It is considered a "purist" genre and tends to exclude anything which includes action elements beyond a mini game.

Because they put little pressure on the player in the form of action-based challenges or time constraints, adventure games have had the unique ability to appeal to people who do not normally play video games. The genre peaked in popularity with the 1993 release of *Myst*, the best-selling PC game of all time up to that point. It had four proper sequels, but none managed to experience the same level of success. The success of *Myst* also inspired many others to create similar games with first person perspectives, surreal environments and minimal or no dialogue, but these neither recaptured the success of *Myst* nor of earlier personality-driven adventures.

- Role-playing video games draw their gameplay from traditional role-playing games like *Dungeons & Dragons*. Most of these games cast the player in the role of one or more "adventurers" who specialize in specific skill sets (such as melee combat or casting magic spells) while progressing through a predetermined storyline. Many involve manoeuvring these character(s) through an overworld, usually populated with monsters, that allows access to more important game locations, such as towns, dungeons, and castles.

Since the emergence of affordable home computers coincided with the popularity of paper and pencil role-playing games, this genre was one of the first in video games and

continues to be popular today. Gameplay elements strongly associated with RPGs, such as statistical character development through the acquisition of experience points, have been widely adapted to other genres such as action-adventure games.

Though nearly all of the early entries in the genre were turn-based games, many modern role-playing games progress in real-time. Thus, the genre has followed the strategy game's trend of moving from turn-based to real-time combat.

7. Write short notes on following (any four):-

(a) Symmetric and Asymmetric Interaction. [5 Marks]

There are two main types of interaction in the matrix: symmetric and asymmetric. Symmetric interactions are the same both ways. For example, the behavior of the ball and the bat are not different depending on how we consider the collision. That is, if we say that the ball collides with the bat, we would expect exactly the same results to occur as if we had said the bat collides with the ball. The semantics do not matter.Symmetric interactions are shown as squares in the matrix.

Asymmetric interactions are shown in the matrix as a square split into two triangles. An asymmetric interaction is different depending on the direction. In this case, the semantics do matter. Each triangle represents one direction of the interaction. Taking the solitary case from the *Pong* matrix as an example, we could say that a goal causes the score to increment by one, but the score incrementing by one does not cause a goal to occur—cause-effect, not effect-cause. (There'll be no breaking the laws of causality in this book!) The matrix allows us to perform a visual check on our interactions. We can check that they are what we would expect, and we can see if we have missed any or made any errors. We may be able to spot unexpected chain reactions (the sort of things that in some cases will enhance a game, but in other cases will render it virtually unplayable). These things will be picked up in play-testing, but the sooner it is spotted the cheaper it is to fix.

(b) Audio Formats. [5 Marks]

An audio file format is a file format for storing digital audio data on a computer system. The bit layout of the audio data (excluding metadata) is called the audio coding format and can be uncompressed, or compressed to reduce the file size, often using lossy compression. The data can be a

raw bitstream in an audio coding format, but it is usually embedded in a container format or an audio data format with defined storage layer.

There are three major groups of audio file formats:

Uncompressed audio formats, such as WAV, AIFF, AU or raw header-less PCM; Formats with lossless compression, such as FLAC, Monkey's Audio (filename extension .ape), WavPack (filename extension .wv), TTA, ATRAC Advanced Lossless, ALAC (filename extension .m4a), MPEG-4 SLS, MPEG-4 ALS, MPEG-4 DST, Windows Media Audio Lossless (WMA Lossless), and Shorten (SHN).

Formats with lossy compression, such as Opus, MP3, Vorbis, Musepack, AAC, ATRAC and Windows Media Audio Lossy (WMA lossy).

- Uncompressed audio format

One major uncompressed audio format, LPCM, is the same variety of PCM as used in Compact Disc Digital Audio and is the format most commonly accepted by low level audio APIs and D/A converter hardware. Although LPCM can be stored on a computer as a raw audio format, it is usually stored in a .wav file on Windows or in a .aiff file on Mac OS. The AIFF format is based on the Interchange File Format (IFF), and the WAV format is based on the similar Resource Interchange File Format (RIFF). WAV and AIFF are not inherently lossless; they're designed to store a wide variety of audio formats, lossless and lossy; they just add a small, metadata-containing header before the audio data to declare the format of the audio data, such as LPCM with a particular sample rate, bit depth, endianness and number of channels. Since WAV and AIFF are widely supported and can store LPCM,

- Lossless compressed audio format

A lossless compressed format stores data in less space without losing any information. The original, uncompressed data can be recreated from the compressed version.

Uncompressed audio formats encode both sound and silence with the same number of bits per unit of time. Encoding an uncompressed minute of absolute silence produces a file of the same size as encoding an uncompressed minute of music. In a lossless compressed format, however, the music would occupy a smaller file than an uncompressed format and the silence would take up almost no space at all.

- Lossy compressed audio format

Lossy compression enables even greater reductions in file size by removing some of the audio information and simplifying the data. This of course results in a reduction in audio quality, but a variety of techniques are used, mainly by exploiting psychoacoustics, to remove the parts of the sound that have the least effect on perceived quality, and to minimize the amount of audible noise added during the process. The popular MP3 format is probably the best-known example, but the AAC format found on the iTunes Music Store is also common. Most formats offer a range of degrees of compression, generally measured in bit rate. The lower the rate, the smaller the file and the more significant the quality loss

(c) Game engine. [5 Marks]

A game engine is a software framework designed for the creation and development of video games. Developers use them to create games for consoles, mobile devices and personal computers. The core functionality typically provided by a game engine includes a rendering engine ("renderer") for 2D or 3D graphics, a physics engine or collision detection (and collision response), sound, scripting, animation, artificial intelligence, networking, streaming, memory management, threading, localization support, and a scene graph.

The process of game development is often economized, in large part, by reusing/adapting the same game engine to create different games or to make it easier to "port" games to multiple platforms. Other middleware solutions, game engines usually provide platform abstraction, allowing the same game to be run on various platforms including game consoles and personal computers with few, if any, changes made to the game source code.

Often, game engines are designed with a component-based architecture that allows specific systems in the engine to be replaced or extended with more specialized (and often more expensive) game middleware components such as Havok for physics, Miles Sound System for sound, or Bink for Video. Some game engines such as RenderWare are even designed as a series of loosely connected game middleware components that can be selectively combined to create a custom engine, instead of the more common approach of extending or customizing a flexible integrated solution. However extensibility is achieved, it remains a high priority for game engines due to the wide variety of uses for which they are applied.

Despite the specificity of the name, game engines are often used for other kinds of interactive applications with real-time graphical needs such as marketing demos, architectural visualizations, training simulations, and modeling environments.

Some game engines only provide real-time 3D rendering capabilities instead of the wide range of functionality needed by games. These engines rely upon the game developer to implement the rest of this functionality or assemble it from other game middleware components. These types of engines are generally referred to as a "graphics engine," "rendering engine," or "3D engine" instead of the more encompassing term "game engine."

This terminology is inconsistently used as many full-featured 3D game engines are referred to simply as "3D engines." A few examples of graphics engines are: Crystal Space, Genesis3D, Irrlicht, OGRE, RealmForge, Truevision3D, and Vision Engine. Modern game or graphics engines generally provide a scene graph, which is an object-oriented representation of the 3D game world which often simplifies game design and can be used for more efficient rendering of vast virtual worlds.

As technology ages, the components of an engine may become outdated or insufficient for the requirements of a given project. Since the complexity of programming an entirely new engine may result in unwanted delays (or necessitate that the project be completely restarted), a development team may elect to update their existing engine with newer functionality or components

(d) Game documentation. [5 Marks]

A game design document (often abbreviated GDD) is a highly descriptive living design document of the design for a video game. A GDD is created and edited by the development team and it is primarily used in the video game industry to organize efforts within a development team. The document is created by the development team as result of collaboration between their designers, artists and programmers as a guiding vision which is used throughout the game development process.

When a game is commissioned by a game publisher to the development team, the document must be created by the development team and it is often attached to the agreement between publisher and developer; the developer has to adhere to the GDD during game development process.

Game development, production, or design is a process that starts from an idea or concept. Often the idea is based on a modification of an existing game concept. The game idea may fall within one or several genres. Designers often experiment with different combinations of genres.

Game designer usually produces initial game proposal document, that contains the concept, gameplay, feature list, setting and story, target audience, requirements and schedule, staff and budget estimates. Different companies have different formal procedures and philosophies regarding game design and development. There is no standardized development method; however commonalities exist.

Game development is undertaken by a game developer—ranging from an individual to a large company. There can be independent or publisher-owned studios. Independent developers rely on financial support from a game publisher. They usually have to develop a game from concept to prototype without external funding. The formal game proposal is then submitted to publishers, who may finance the game development from several months to years.

The publisher would retain exclusive rights to distribute and market the game and would often own the intellectual property rights for the game franchise. Publisher's company may also own the developer's company, or it may have internal development studio(s). Generally the publisher is the one who owns the game's intellectual property rights.

All but the smallest developer companies work on several titles at once. This is necessary because of the time taken between shipping a game and receiving royalty payments, which may be between 6 to 18 months. Small companies may structure contracts, ask for advances on royalties, use shareware distribution, employ part-time workers and use other methods to meet payroll demands.

Console manufacturers, such as Microsoft, Nintendo, or Sony, have a standard set of technical requirements that a game must conform to in order to be approved. Additionally, the game concept must be approved by the manufacturer, who may refuse to approve certain titles.

Most modern PC or console games take from one to three years to complete where as a mobile game can be developed in a few months.[37] The length of development is influenced by a number of factors, such as genre, scale, development platform and amount of assets.

(e) Game play. [5 Marks]

Gameplay is the specific way in which players interact with a game, and in particular with video games. Gameplay is the pattern defined through the game rules, connection between player and the game, challenges and overcoming them, plot and player's connection with it. Video game gameplay is distinct from graphics and audio elements.

Interaction is a kind of action that occurs as two or more objects have an effect upon one another. The idea of a two-way effect is essential in the concept of interaction, as opposed to a one-way causal effect. A closely related term is interconnectivity, which deals with the interactions of interactions within systems: combinations of many simple interactions can lead to surprising emergent phenomena. *Interaction* has different tailored meanings in various sciences. Changes can also involve interaction.

Casual examples of interaction outside science include:

- Communication of any sort, for example two or more people talking to each other, or communication among groups, organizations, nations or states: trade, migration, foreign relations, transportation

- The feedback during the operation of machines such as a computer or tool, for example the interaction between a driver and the position of his or her car on the road: by steering the driver influences this position, by observation this information returns to the driver.

(f) Scene nodes. [5 Marks]

A scene graph is a set of tree data structures where every item has zero or one parent, and each item is either a "leaf" with zero sub-items or a "branch" with zero or more sub-items.

Each item in the scene graph is called a Node. Branch nodes are of type Parent, whose concrete subclasses are Group, Region, and Control, or subclasses thereof.

Leaf nodes are classes such as Rectangle, Text, ImageView, MediaView, or other such leaf classes which cannot have children. Only a single node within each scene graph tree will have no parent, which is referred to as the "root" node.

There may be several trees in the scene graph. Some trees may be part of a Scene, in which case they are eligible to be displayed. Other trees might not be part of any Scene.

A node may occur at most once anywhere in the scene graph. Specifically, a node must appear no more than once in all of the following: as the root node of a Scene, the children ObservableList of a Parent, or as the clip of a Node.

The scene graph must not have cycles. A cycle would exist if a node is an ancestor of itself in the tree, considering the Group content ObservableList, Parent children ObservableList, and Node clip relationships mentioned above.

If a program adds a child node to a Parent (including Group, Region, etc) and that node is already a child of a different Parent or the root of a Scene, the node is automatically (and silently) removed from its former parent. If a program attempts to modify the scene graph in any other way that violates the above rules, an exception is thrown, the modification attempt is ignored and the scene graph is restored to its previous state.

It is possible to rearrange the structure of the scene graph, for example, to move a subtree from one location in the scene graph to another. In order to do this, one would normally remove the subtree from its old location before inserting it at the new location. However, the subtree will be automatically removed as described above if the application doesn't explicitly remove it.

Node objects may be constructed and modified on any thread as long they are not yet attached to a Scene. An application must attach nodes to a Scene, and modify nodes that are already attached to a Scene, on the JavaFX Application Thread.

- String ID

Each node in the scene graph can be given a unique id. This id is much like the "id" attribute of an HTML tag in that it is up to the designer and developer to ensure that the id is unique within the scene graph. A convenience function called lookup(String) can be used to find a node with a unique id within the scene graph, or within a subtree of the scene graph. The id can also be used identify nodes for applying styles; see the CSS section below.

- Coordinate System

The Node class defines a traditional computer graphics "local" coordinate system in which the x axis increases to the right and the y axis increases downwards. The concrete node classes for shapes provide variables for defining the geometry and location of the shape within this local coordinate space.

For example, Rectangle provides x, y, width, height variables while Circle provides centerX, centerY, and radius.

At the device pixel level, integer coordinates map onto the corners and cracks between the pixels and the centers of the pixels appear at the midpoints between integer pixel locations. Because all coordinate values are specified with floating point numbers, coordinates can precisely point to these corners (when the floating point values have exact integer values) or to any location on the pixel. For example, a coordinate of (0.5, 0.5) would point to the center of the upper left pixel on the Stage. Similarly, a rectangle at (0, 0) with dimensions of 10 by 10 would span from the upper left corner of the upper left pixel on the Stage to the lower right corner of the 10th pixel on the 10th scanline. The pixel center of the last pixel inside that rectangle would be at the coordinates (9.5, 9.5).

In practice, most nodes have transformations applied to their coordinate system as mentioned below. As a result, the information above describing the alignment of device coordinates to the pixel grid is relative to the transformed coordinates, not the local coordinates of the nodes. The Shape class describes some additional important context-specific information about coordinate mapping and how it can affect rendering.

- Transformations

Any Node can have transformations applied to it. These include translation, rotation,scaling, or shearing.

A translation transformation is one which shifts the origin of the node's coordinate space along either the x or y axis. For example, if you create a Rectangle which is drawn at the origin (x=0, y=0) and has a width of 100 and a height of 50, and then apply a Translate with a shift of 10 along the x axis (x=10), then the rectangle will appear drawn at (x=10, y=0) and remain 100 points wide and 50 tall. Note that the origin was shifted, not the x variable of the rectangle.

A common node transform is a translation by an integer distance, most often used to lay out nodes on the stage. Such integer translations maintain the device pixel mapping so that local coordinates that are integers still map to the cracks between pixels.

A rotation transformation is one which rotates the coordinate space of the node about a specified "pivot" point, causing the node to appear rotated. For example, if you create a Rectangle which is drawn at the origin (x=0, y=0) and has a width of 100 and height of 30 and

you apply a Rotate with a 90 degree rotation (angle=90) and a pivot at the origin (pivotX=0, pivotY=0), then the rectangle will be drawn as if its x and y were zero but its height was 100 and its width -30.

That is, it is as if a pin is being stuck at the top left corner and the rectangle is rotating 90 degrees clockwise around that pin. If the pivot point is instead placed in the center of the rectangle (at point x=50, y=15) then the rectangle will instead appear to rotate about its center.

Mr. Jayant Rohankar
Subject I/C

HoD [Info. Tech.]