Affiliated to
VISVESVARAYA TECHNOLOGICAL UNIVERSITY
BELGAUM

JSS MAHAVIDYAPEETHA
SRI JAYACHAMARAJENDRA COLLEGE OF ENGINEERING
MYSORE-570006
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Seminar report on

3D Game Development and Design

Name	USN	Roll number	Section
Bharath S N	4JC09CS014	07	B

Under Guidance of
Mr. DIVAKARA N
Asst. Professor

J S S Mahavidyapeetha

Sri Jayachamarajendra College of Engineering (SJCE), Mysore – 570 006

An Autonomous Institute Affiliated to

Visvesvaraya Technological University, Belgaum

Certificate

Certified that this is a bonafide record of the seminar entitled

3D Game Development and Design

By

Bharath S N

8th semester, Computer Science and Engineering in the year 2013 as partial fulfillment of the requirements to award Degree of **Bachelor of Engineering in Computer Science and Engineering under Visveshwaraya Technological University (VTU)**. It is certified that all corrections / suggestions indicated during seminar presentation have been incorporated in the report. The seminar report has been approved as it satisfies the academic requirements in respect of seminar presentation prescribed for the Bachelor of Engineering degree.

Seminar Guide	Panel Member	Head of the Department
Mr. Divakara N	**Smt. Manjula S**	**Dr. C.N.Ravikumar**
Asst. Professor	**Asst. Professor**	**Professor & Head**

Place: Mysore

Date: 25th March 2013

ACKNOWLEDGEMENT

The satisfaction and pleasure that accompany the gain of experience would be incomplete without the mention of the people who made it possible.

First and foremost I ought to pay due regards to this renowned institution, which provided us a platform and an opportunity for carrying out this project work.

My sincere regards to **Dr. B. G. Sangameshwara**, **Principal of S.J.C.E** for providing us an opportunity to enhance our knowledge by working on this project. I also thank **Dr. C.N. Ravikumar**, **Professor and Head of department** of **Computer science and Engineering** for the implementation of plan of the project in our curriculum to enhance the knowledge of the students.

I take this opportunity to express my profound gratitude and deep regards to my guide **Asst. prof. Divakara N** for his encouragement and guidance in carrying out this seminar.

Contents

Abstract

A video game is an electronic game that involves human interaction with a user interface to generate visual feedback on a video device. This document gives the overview of the 3D game development. Here we mainly concentrate on how game works under the hood.

Ogre is an acronym for Object-oriented 3D Graphics Rendering Engine. It is an open source engine mainly involves designing and developing game right from the scratch. Scene managers are the basic entity which is a key to render the objects in the scene on to the display. BSP scene manager is an obsolete approach for rendering but still in use. BSP is mainly used to render the objects present in a closed space. For eg. House, building , etc.. Octree scene manager is one among the advanced rendering algorithm and much more efficient than the BSP in the matter of time, space, visible surface determination etc..

This document also concentrates on the advanced set of tools available for game development and their usage. Gives introduction about the Game Engines having GUI. Finally at the end we'll be glancing at the shader programming and its importance. Shaders are needed to create custom environment and runs on GPU.

1. INTRODUCTION

1.1 3D Computer graphics :-

3D computer graphics (in short computer graphics) are graphics that use a three-dimensional representation of geometric data (often Cartesian) that is stored in the computer for the purposes of performing calculations and rendering 2D images. Computer graphics rely on many of the same algorithms as 2D computer vector graphics in the wire-frame model and 2D computer raster graphics in the final rendered display. In computer graphics software, the distinction between 2D and 3D is occasionally blurred; 2D applications may use 3D techniques to achieve effects such as lighting, and 3D may use 2D rendering techniques.

Computer graphics are often referred to as 3D models. Apart from the rendered graphic, the model is contained within the graphical data file. However, there are differences. A 3D model is the mathematical representation of any three-dimensional object. A model is not technically a graphic until it is displayed. Due to 3D printing, 3D models are not confined to virtual space. A model can be displayed visually as a two-dimensional image through a process called 3D rendering, or used in non-graphical computer simulations and calculations.

1.2 3 basic steps in game development :-

- **3D modeling :-** The model describes the process of forming the shape of an object. The two most common sources of 3D models are those that an artist or engineer originates on the computer with some kind of 3D modeling tool, and models scanned into a computer from real-world objects. Models can also be produced procedurally or via physical simulation. Basically, a 3D model is formed from points called vertices that define the shape and form polygons.
- **Layout and Animation :-** Before rendering into an image, objects must be placed in a scene. This defines spatial relationships between objects, including location and size. Animation refers to the temporal description of

an object, i.e., how it moves and deforms over time. Popular methods include key framing, inverse kinematics, and motion capture. These techniques are often used in combination. As with modeling, physical simulation also specifies motion.

- **Rendering: -** Rendering converts a model into an image either by simulating light transport to get photo-realistic images, or by applying some kind of style as in non-photorealistic rendering. The two basic operations in realistic rendering are transport (how much light gets from one place to another) and scattering (how surfaces interact with light). This step is usually performed using 3D computer graphics software or a 3D graphics API.

1.3 Game Engine: -

A ***game engine*** is a system / software product designed for the creation and development of video games. The term “game engine” arose in the mid-1990s in reference to first-person shooter (FPS) games like the insanely popular Doom by id Software. Doom was architected with a reasonably well-defined separation between its core software components and the art assets, game worlds, and rules of play that comprised the player’s gaming experience. The line between a game and its engine is oft en blurry. Some engines make a reasonably clear distinction, while others make almost no attempt to separate the two. In one game, the rendering code might “know” specifically how to draw an arc. In another game, the rendering engine might provide general-purpose material and shading facilities, and “orc-ness” might be defined entirely in data.

The leading game engines provide a software framework that developers use to create games for video game consoles 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.

Arguably a data-driven architecture is what differentiates a game engine from a piece of software that is a game but not an engine. When a game contains hard coded logic or game rules, or employs special-case code to render specific types of game objects, it becomes difficult or impossible to reuse that software to make a different game. We should probably reserve the term “game engine” for software that is extensible and can be used as the foundation for many different games without major modification.

1.4 Literature review: -

Before game engines, games were typically written as singular entities: a game for the Atari 2600, for example, had to be designed from the bottom up to make optimal use of the display hardware—this core display routine is today called the kernel by retro developers. Other platforms had more leeway, but even when the display was not a concern, memory constraints usually sabotaged attempts to create the data-heavy design that an engine needs. Thus most game designs through the 1980s were designed through a hard-coded rule set with a small number of levels and graphics data. Since the golden age of arcade video games, it became common for video game companies to develop in-house game engines for use with first-party software.

While third-party game engines were not common up until the rise of 3D computer graphics in the 1990s, there were several 2D game creation systems produced in the 1980s for independent video game development. These include Pinball Construction Set (1983), ASCII's War Game Construction Kit (1983),[5] Thunder Force Construction (1984),[6] Adventure Construction Set (1984), Garry Kitchen's GameMaker (1985), Wargame Construction Set (1986), Shoot'Em-Up Construction Kit (1987), Arcade Game Construction Kit (1988), and most popularly ASCII's RPG Maker engines from 1988 onwards.

Later games, such as id Software's Quake III Arena and Epic Games's 1998 Unreal were designed with this approach in mind, with the engine and content developed separately. The practice of licensing such technology has proved to be a useful auxiliary revenue stream for some game developers, as a one license for a high-end commercial game engine can range from US$10,000 to millions of dollars, and the number of licensees can reach several dozen companies, as seen with the Unreal Engine. Modern game engines are some of the most complex applications written, often featuring dozens of finely tuned systems interacting to ensure a precisely controlled user experience.

First-person shooter games remain the predominant users of third-party game engines, but they are now also being used in other genres. For example, the role-playing video game The Elder Scrolls III: Morrowind and the MMORPG Dark Age of Camelot are based on the Gamebryo engine, and the MMORPG Lineage II is based on the Unreal Engine. Game engines are used for games originally developed for home consoles as well; for example, the RenderWare engine is used in the Grand Theft Auto and Burnout franchises.

1.5 Game Genres: -

- **First-Person Shooters (FPS): -** FPS is a video game genre centered on gun and projectile weapon-based combat through a first-person perspective; that is, the player experiences the action through the eyes of the protagonist. The first-person shooter shares common traits with other shooter games, which in turn fall under the heading action game. From the genre's inception, advanced 3D or pseudo-3D graphics have challenged hardware development, and multiplayer gaming has been integral. Eg., Call of duty, Crysis, counter strike. etc.
- **Platformers and Other Third-Person Games: -** A platform game (or platformer) is a video game which requires an avatar to jump to and from suspended platforms or over obstacles (jumping puzzles). The player must control these jumps to avoid the avatar falling from platforms or missing necessary jumps. Platform games originated in the early 1980s, and 3D successors were popularized in the mid-1990s. The term itself describes games where jumping on platforms is an integral part of the gameplay, and came into use some time after the genre had been established, but no later than 1983. Eg., Space Panic, Donkey Kong, etc.
- **Fighting Games: -** Fighting game is a video game genre where the player controls an on-screen character and engages in close combat with an opponent. These characters tend to be of equal power and fight matches consisting of several rounds, which take place in an arena. Players must

master techniques such as blocking, counter-attacking, and chaining together sequences of attacks known as "combos". Eg., Fight Night Round 3, mortal combat , etc.

- **Racing Games: -** A racing video game is a genre of video games, either in the first-person or third-person perspective, in which the player partakes in a racing competition with any type of land, air, or sea vehicles. They may be based on anything from real-world racing leagues to entirely fantastical settings. In general, they can be distributed along a spectrum anywhere between hardcore simulations, and simpler arcade racing games. Eg., Need for speed ,blur, etc.
- **Real-Time Strategy (RTS): -** In this genre the participants position and maneuver units and structures under their control to secure areas of the map and/or destroy their opponents' assets. In a typical RTS, it is possible to create additional units and structures during the course of a game. This is generally limited by a requirement to expend accumulated resources. These resources are in turn garnered by controlling special points on the map and/or possessing certain types of units and structures devoted to this purpose. Eg., The Building of a Dynasty, Age of empires, etc.
- **Massively Multiplayer Online Games (MMOG): -** A massively multiplayer online game (also called MMO and MMOG) is a multiplayer video game which is capable of supporting large numbers of players simultaneously. By necessity, they are played on the Internet. Many games have at least one persistent world, however others just have large numbers of players competing at once in one form or another without any lasting effect to the world at all. Eg., warface, Neverwinter Nights, EverQuest, World of Warcraft , and Star Wars Galaxies, etc.

1.6 List of Top 10 Game Engines: -

1. ***RAGE (Rock Star Advanced Game Engine) :-*** GTA Series
2. ***CryENGINE 3 :-*** Crysis series, warface.
3. ***Naughty Dog Game Engine*** :- Uncharted: Drake's Fortune
4. ***The Dead Engine :-*** Dead Space
5. ***Unreal Engine :-*** Mass Effect Series, Call Of Duty Black ops
6. ***Avalanche Engine :-*** The Hunter

7. ***IW (Infinity Ward) Engine :-*** Call of Duty MW1 and 2
8. ***Anvil Engine :-*** Assassin's Creed1 & 2, Prince of Persia
9. ***EGO Engine :-*** Dirt
10. ***Geo-Mod Engine :-*** Red Faction: Guerrilla

2. OGRE GAME ENGINE

2.1 Introduction: -

OGRE is an acronym for Object-Oriented Graphics Rendering Engine. Ogre is a scene-oriented, real-time, flexible 3D rendering engine written in C++ designed to make it easier and intuitive for developers to produce applications utilizing hardware-accelerated 3D graphics. The class library abstracts the details of using the underlying system libraries like Direct3D and OpenGL and provides an interface based on world objects and other high level classes.

OGRE has received multi-platform support and currently supports Linux, Windows, Mac OSX, Windows Phone 8, iOS and Android. Its main purpose is to provide a general solution for graphics rendering. Though it also comes with other facilities like vector and matrix classes, memory handling, etc., they are considered supplemental. It is not an all-in-one solution in terms of game development or simulation as it doesn't provide audio or physics support.

2.2 System Requirements: -

- **NVidia:** Geforce2 or higher required, Geforce 4(non-mx) or higher recommended.
- **ATI:** Radeon 7500 or higher required, Radeon 9600 or higher recommended.
- **OpenGL :** An OpenGL driver exposes a certain core version and a set of extensions. The OpenGL core version defines the set of base capabilities, while the extensions supply external features that can be used by applications and games to improve graphical quality or increase

performance. To work at all, OGRE requires a base OpenGL version of 1.2.1.

- You need a compiler to compile the applications. Your computer should have a graphic card with 3D capabilities. It would be best if the graphic card supports DirectX 9.0.
- Processor should be fitting for your graphics card, so at least around 500Mzh would be my guess. Just guess though. RAM depends on your project. Ogre demos usually use around 50 - 80 MByte memory

2.3 Features of Ogre: -

- OGRE has an object-oriented design with a plugin architecture
- OGRE is a scene graph based engine,
- It supports variety of scene managers
- Platform & 3D API support
- Material / Shader support
- Meshes , Animation, Special Effects, etc.

2.4 What is a Scene Manager ? : -

1. **What is a scene…? : -** A scene is an abstract representation of what is shown in a virtual world. It consists of static geometry such as terrain building interiors, models such as trees, chairs or monsters, light sources that illuminate the scene. It also consists of cameras that view the scene.

2. **Ogre supports the following set of scene managers: -**
   - i. Generic/Default SceneManager
   - ii. Octree SceneManager
   - iii. BSP(Binary Space Partitioning) SceneManager
   - iv. PCZ (Portal Connected Zone) SceneManager

3. **What does a SceneManager do …? : -** It helps the Ogre Engine to monitor and maintain all the assets present in the current scene. SceneManager is capable of handling scene nodes, entities, lights, Particles, or a lot of other object types. The SceneManager object is under the control of the Root object. The Root object/node is the father of all the assets present in the scene and of course u can navigate from one node to another by using root object.

2.5 Ogre Scene Managers: -

- **Generic/Default: -** It's the default scene manager given by the Ogre Engine if not specified. Uses Built hierarchy for frustum culling, RayCasting is worst.
- **BSP: -** It is an Old technique but this is intended for use in interior scenes. it is optimized for the sort of geometry that results from intersecting walls and corridors.
- **Octree: -** It is a data structure in which a node can have at most 8 children. It will quickly cull entire regions of the world(In the World space).That is if a parent region is not visible to the camera then there is no need to check the children of that node. RayCasting is efficient.
- **PCZ: -**The Portal Connected Zone manager allows you to define zones in the world and portals which connect them. It's harder to set up because this isn't automatic, you need to place zones and portals Optimized for interior scenes. Compatible with numerous level editing tools.

2.5.1 BSP Scene Manager: -

Binary space partitioning is a generic process of recursively dividing a scene into two until the partitioning satisfies one or more requirements. When used in computer graphics to render scenes composed of planar polygons, the partitioning planes are frequently (but not always) chosen to coincide with the planes defined by polygons in the scene. The specific choice of partitioning plane and criterion for terminating the partitioning process varies depending on the purpose of the BSP tree. For example, in computer graphics rendering, the scene is divided until each node of the BSP tree contains only polygons that can render in arbitrary order. When backface culling is used, each node therefore contains a convex set of polygons, whereas when rendering double-sided polygons, each node of the BSP tree contains only polygons in a single plane. In collision detection or ray tracing, a scene may be divided up into primitives on which collision or ray intersection tests are straightforward.

- Introduced by **Fuchs**, **Kedem**, and **Naylor** around 1980.
- It uses the painter's algorithm for creating and sorting the objects/nodes in the scene.
- Painter's algorithm allows to order the objects stored at the leaves of a BSP tree in back-to-front order from the viewpoint
- Enhances the rendering of static scenes.
- It uses z-buffers for rendering the objects, then it checks the z-value for each and every polygon so that closest is drawn last.
- BSP Algorithm involves 2 steps
  - Generating tree
  - Traversing the generated tree

2.5.2 BSP algorithm(Existing solution): -

The recursive algorithm for construction of a BSP tree from that list of polygons

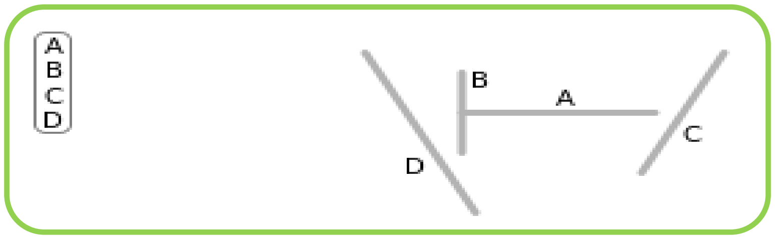
1. Choose a polygon *P* from the list.
2. Make a node *N* in the BSP tree, and add *P* to the list of polygons at that node.
3. For each other polygon in the list:
   - If that polygon is wholly in front of the plane containing *P*, move that polygon to the list of nodes in front of *P*.

- If that polygon is wholly behind the plane containing *P*, move that polygon to the list of nodes behind *P*.
- If that polygon is intersected by the plane containing *P*, split it into two polygons and move them to the respective lists of polygons behind and in front of *P*.
- If that polygon lies in the plane containing *P*, add it to the list of polygons at node *N*.

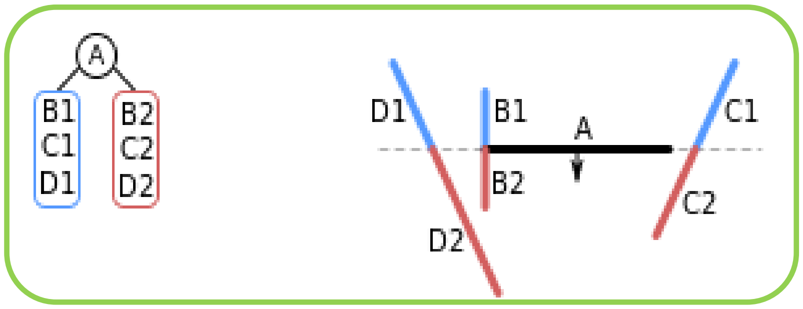
4. Apply this algorithm to the list of polygons in front of *P*.
5. Apply this algorithm to the list of polygons behind *P*.

Pictorial representation: -

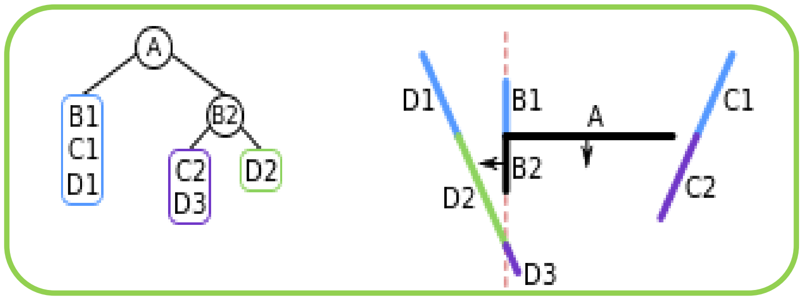
Let us assume A,B,C and D are polygons and we are viewing the space from the top.Rounded rectangle is the List.

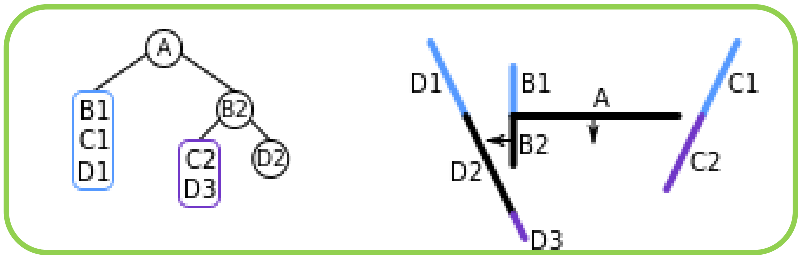


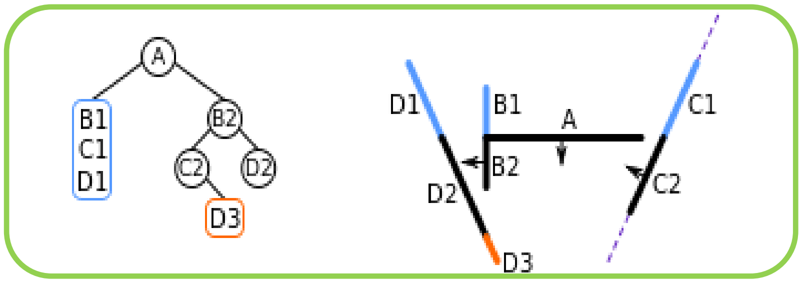
Choose a line, A, from the list and add it to a node. split the remaining lines in the list into those in front of A and those behind.

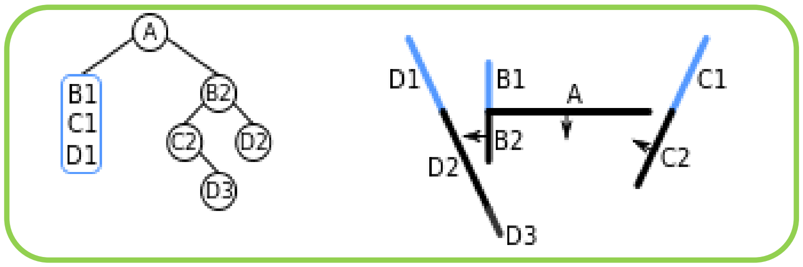


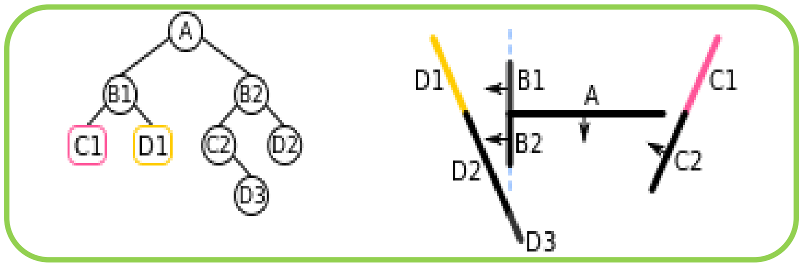
Repeat the process for each and every sub polygons as follows.

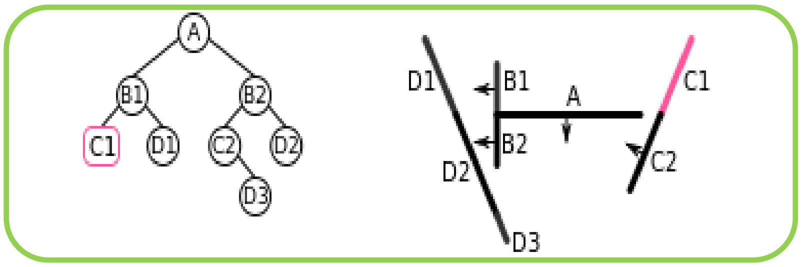

A
B1
C1
D1
B2
C2
D3
D2
D1
B1
A
C1
B2
C2
D2
D3

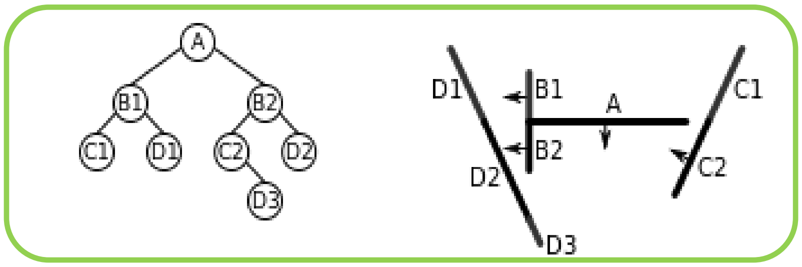

A
B1
C1
D1
B2
C2
D3
D2
D1
B1
A
C1
B2
C2
D2
D3


A
B1
C1
D1
B2
C2
D2
D3
D1
B1
A
C1
B2
C2
D2
D3


A
B1
C1
D1
B2
C2
D2
D3
D1
B1
A
C1
B2
C2
D2
D3



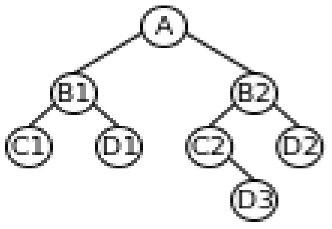


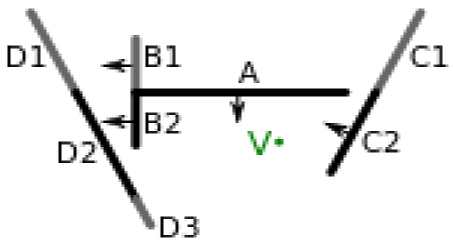


Traversing the tree…

- If the current node is a leaf node, render the polygons at the current node.
- Otherwise, if the viewing location *V* is in front of the current node:
  - Render the child BSP tree containing polygons behind the current node
  - Render the polygons at the current node
  - Render the child BSP tree containing polygons in front of the current node

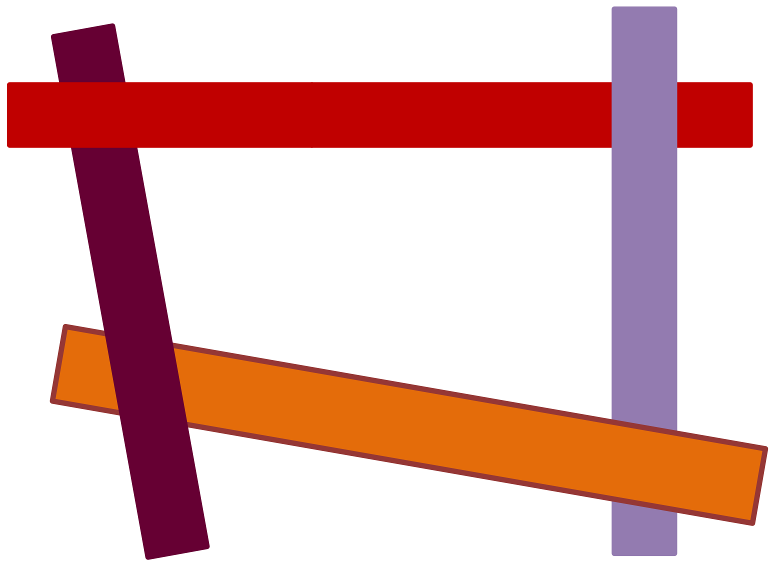
- Otherwise, if the viewing location *V* is behind the current node:
  - Render the child BSP tree containing polygons in front of the current node
  - Render the polygons at the current node
  - Render the child BSP tree containing polygons behind the current node
- Otherwise, the viewing location *V* must be exactly on the plane associated with the current node. Then:
  - Render the child BSP tree containing polygons in front of the current node
  - Render the child BSP tree containing polygons behind the current node





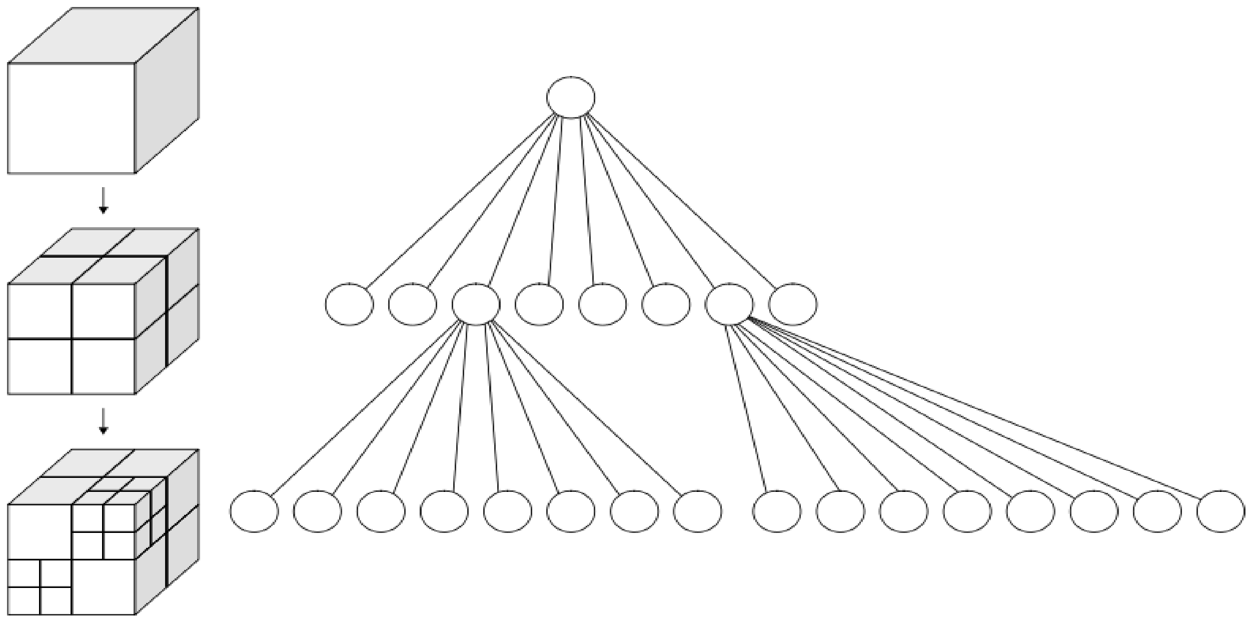
BSP Drawbacks: -

- Polygons will not be drawn correctly if they pass through any other polygon.
- It is difficult and computationally expensive to calculate the order that the polygons should be drawn in for each frame.
- The algorithm cannot handle cases of cyclic overlap as shown in the figure below.
- It does not solve the problem of visible surface determination.
- It consumes more memory space.



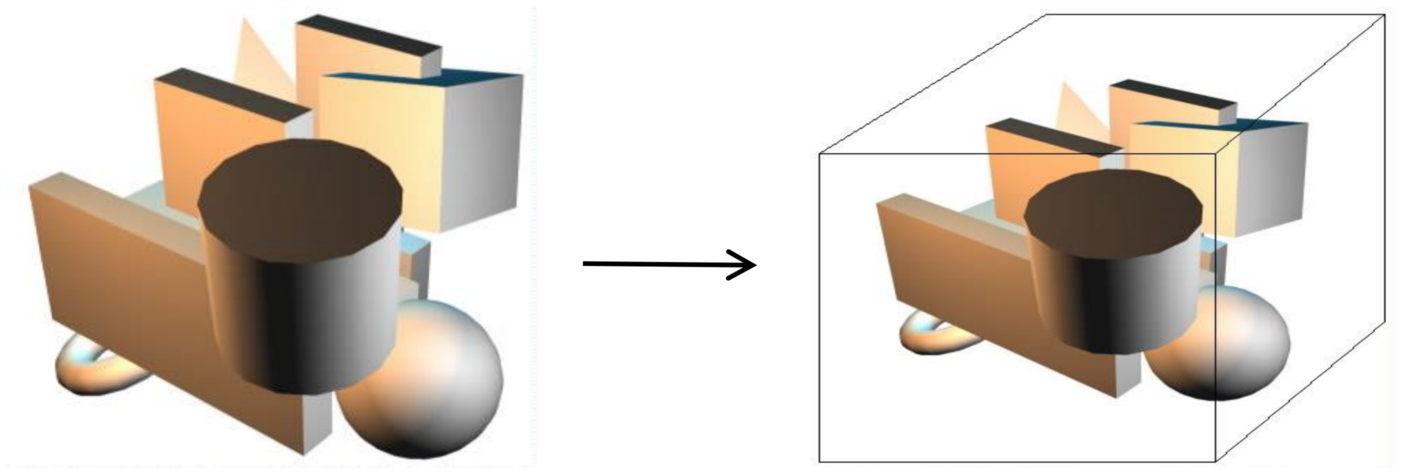
2.5.3 Octree Scene Manager: -

- It is more efficient than the BSP scenemanager
- Octree solves all the problems and drawbacks of BSP
- Just like other trees it has a root, each node has a parent, has a maximum of 8 childern.



Octree Algorithm

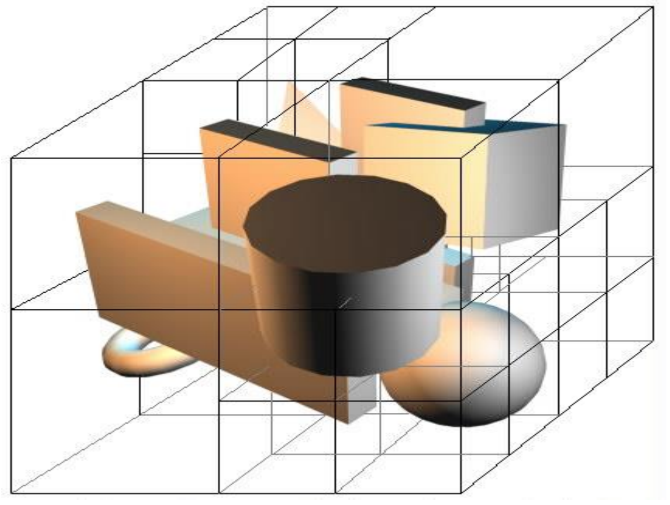
- Each node in an octree subdivides the space it represents into eight octants
- Each node also has a chosen middle point inside this box
- For example :- consider a scene bunch of polygons ramdomly distributed in the space. The construction goes like…



2.5.4 Octree algorithm(advanced solution): -

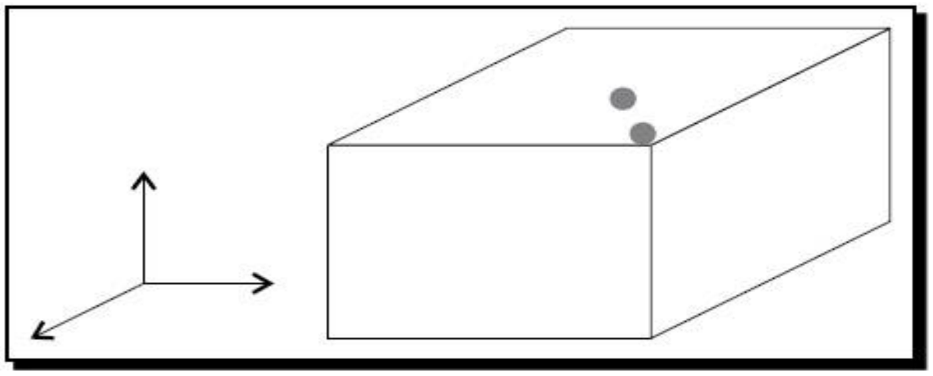
The Octree algorithm generates the final tree containing the objects in the divided space.

- We can the larger cubes as parents and smaller cubes as children.
- There are two ways to stop the subdividing process.
    - The first one is to stop when a cube has some size smaller then a fixed number.
    - The second one is to stop subdividing when the number of polygons in a cube is smaller than some fixed number.

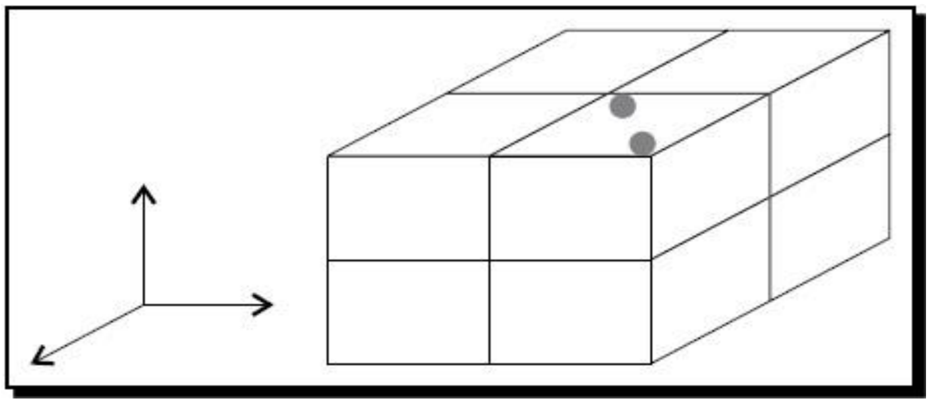


Octree Creation

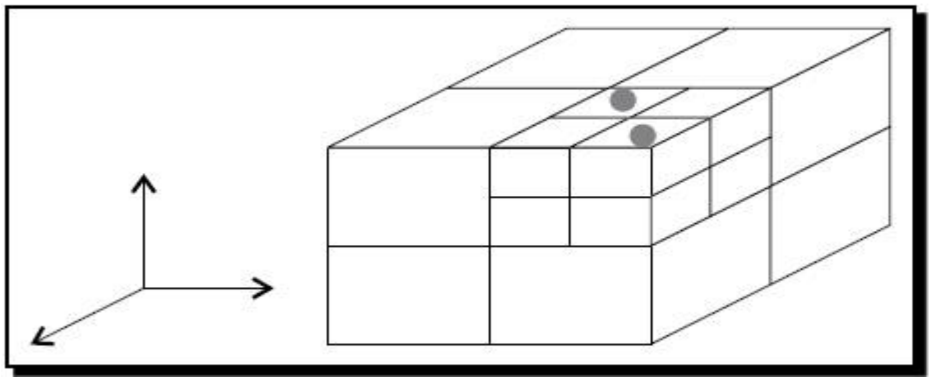
- ✓ Now consider a 3D scene with two objects in it.
- ✓ Enclose those two objects in a cube.



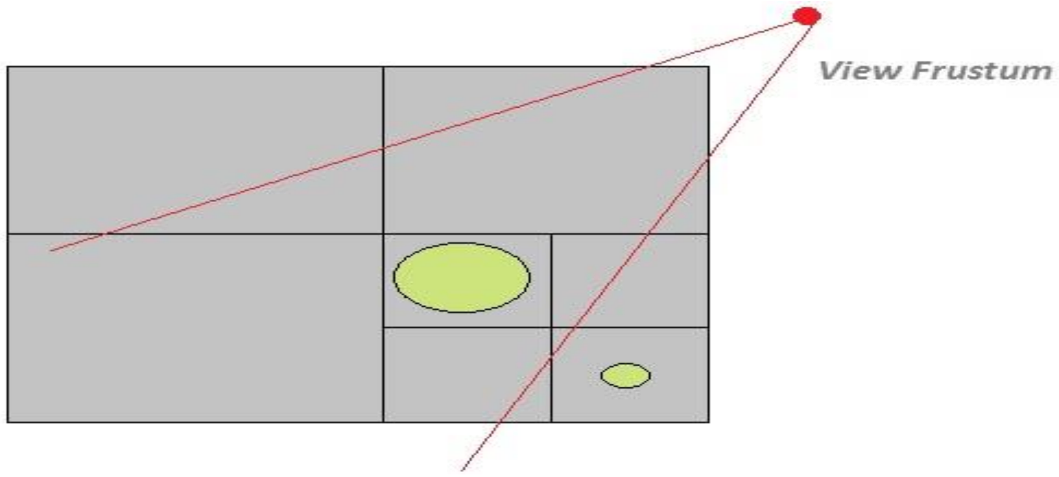
- ✓ Divide the cube at half of its width, height and depth. The resultant 8 cubes can be thought of 8 children of the original cube.



- ✓ Now the two objects of the scene are in the right upper-front cube. The seven other cubes are empty and are therefore leaves. We will divide the cube that contains the two objects again.



- ✓ Each cube has either one or no object enclosed and is a leaf.
- ✓ This property of an Octree makes culling easy and fast.



- ✓ To determine which objects have to be rendered, we just have to check If the view frustum intersects with each and every child the cube or not.
- ✓ If intersects then draw the cube else cull the cube.

2.6 System Design and Implementation: -

BSP.h file

```
#include "SdkSample.h"
#include "FileSystemLayer.h"

using namespace Ogre;
using namespace OgreBites;

class _OgreSampleClassExport Sample_BSP : public SdkSample
{
public:

	Sample_BSP()
	{
		mInfo["Title"] = "BSP";
		mInfo["Description"] = "A demo of the indoor, or BSP (Binary Space Partition) scene 
manager. "
			"Also demonstrates how to load BSP maps from Quake 3.";
		mInfo["Thumbnail"] = "thumb_bsp.png";
		mInfo["Category"] = "Geometry";
	}

	StringVector getRequiredPlugins()
	{
		StringVector names;
		names.push_back("BSP Scene Manager");
		return names;
	}

protected:

	void locateResources()
	{
		ConfigFile cf;
		cf.load(mFSLayer->getConfigFilePath("quakemap.cfg"));
		mArchive = cf.getSetting("Archive");
		mMap = cf.getSetting("Map");

#if OGRE_PLATFORM == OGRE_PLATFORM_APPLE || OGRE_PLATFORM == 
OGRE_PLATFORM_APPLE_IOS

      if (!Ogre::StringUtil::startsWith(mArchive, "/", false))
         mArchive = Ogre::String(Ogre::macBundlePath() + "/" + mArchive);
#endif


	ResourceGroupManager::getSingleton().addResourceLocation(mArchive, "Zip",
			ResourceGroupManager::getSingleton().getWorldResourceGroupName(), true);
	}

	void createSceneManager()
	{
		mSceneMgr = mRoot->createSceneManager("BspSceneManager");	}

	void loadResources()
	{
		mTrayMgr->showLoadingBar(1, 1, 0);

		ResourceGroupManager& rgm = ResourceGroupManager::getSingleton();
		rgm.linkWorldGeometryToResourceGroup(rgm.getWorldResourceGroupName(),
mMap, mSceneMgr);
		rgm.initialiseResourceGroup(rgm.getWorldResourceGroupName());
		rgm.loadResourceGroup(rgm.getWorldResourceGroupName(), false);

		mTrayMgr->hideLoadingBar();
	}

	void unloadResources()
	{
		ResourceGroupManager& rgm = ResourceGroupManager::getSingleton();
		rgm.unloadResourceGroup(rgm.getWorldResourceGroupName());
		rgm.removeResourceLocation(mArchive,
ResourceGroupManager::getSingleton().getWorldResourceGroupName());
	}

	void setupView()
	{
		SdkSample::setupView();

		mCamera->setNearClipDistance(4);
		mCamera->setFarClipDistance(4000);

		ViewPoint vp = mSceneMgr->getSuggestedViewpoint(true);

		mCamera->setFixedYawAxis(true, Vector3::UNIT_Z);
		mCamera->pitch(Degree(90));

		mCamera->setPosition(vp.position);
		mCamera->rotate(vp.orientation);

		mCameraMan->setTopSpeed(350);	}

	String mArchive;
	String mMap;
};
```

BSP.cpp

```
#include "SamplePlugin.h"
#include "BSP.h"

using namespace Ogre;
using namespace OgreBites;

SamplePlugin* sp;
Sample* s;


extern "C" _OgreSampleExport void dllStartPlugin()
{
	s = new Sample_BSP;
	sp = OGRE_NEW SamplePlugin(s->getInfo()["Title"] + " Sample");
	sp->addSample(s);
	Root::getSingleton().installPlugin(sp);
}

extern "C" _OgreSampleExport void dllStopPlugin()
{
	Root::getSingleton().uninstallPlugin(sp);
	OGRE_DELETE sp;
	delete s;
}
```

3. *Advanced Game Engine*

3.1 Game Engines having GUI

- The traditional Ogre engine doesn’t have GUI.
- Developing games from OGRE platform need to start from the scratch.
- Would be time consuming.
- Have to hard code our own customized shaders for texturing.
- Have to plot the co-ordinate in the in the 3D space to placing the objects.

- ➢ One solution for all the problems is to use the game engines having GUI.
- ➢ Drag and drop Environment.
- ➢ Ogre has a GUI enabled engine called “neo-axis”.

3.2 CryENGINE 3

- ✓ The CryENGINE 3 is a game engine which drives the visual actions taking place on the screen.
- ✓ It is a wholly “What U See Is What U Play” technology.
- ✓ Game Engine designed and developed by Crytek.
- ✓ The most prominent tool provided by a game engine is the level. This is done by CryENGINE 3 sandbox.
- ✓ It contains Each and every tools needed for level designing like, assets, models, animations, texturing, Video rendering, Audio subsystem, AI subsystem, material editor, etc…
- ✓ It is specifically designed for first-person shooter genre.
- ✓ Freely available at www.crydev.net.

Not every new computer game needs to start from scratch. A technique that's becoming increasingly common for most developers is to reuse existing game engines. This is where the CryENGINE 3 SDK comes in. The CryENGINE 3 SDK is a game engine which drives the visual actions taking place on the screen. Within this engine are the rules that dictate the way the game world works, and how objects and characters should behave within it. Due to the fact that creating the underlying code for the variety of systems within a game engine is usually very expensive and time consuming, the starting point of working with a game engine thus makes excellent financial sense for most developers. Throughout this book, and for all intents and purposes, we will be referring to the CryENGINE 3 SDK as the engine.

3.3 Tools and Technology used: -

- NVIDIA fx composer
- CryENGINE 3 SDK
- Autodesk 3DS Max
- Autodesk Mudbox
- Adobe Photoshop
- Crazy bump

4. ***Implementation and Results***

4.1 Introduction to Shaders: -

- A shader is a computer program that runs on the graphics processing unit and is used to do shading.
- A programmer can think of the main program as being executed just once on a CPU. But a shader program is executed repeatedly on a GPU.
- Executed for each and every elements of data in a stream.
- Shaders calculate rendering effects on graphics hardware with a high degree of flexibility.
- The material editor auto-generates the shader code during texturing.

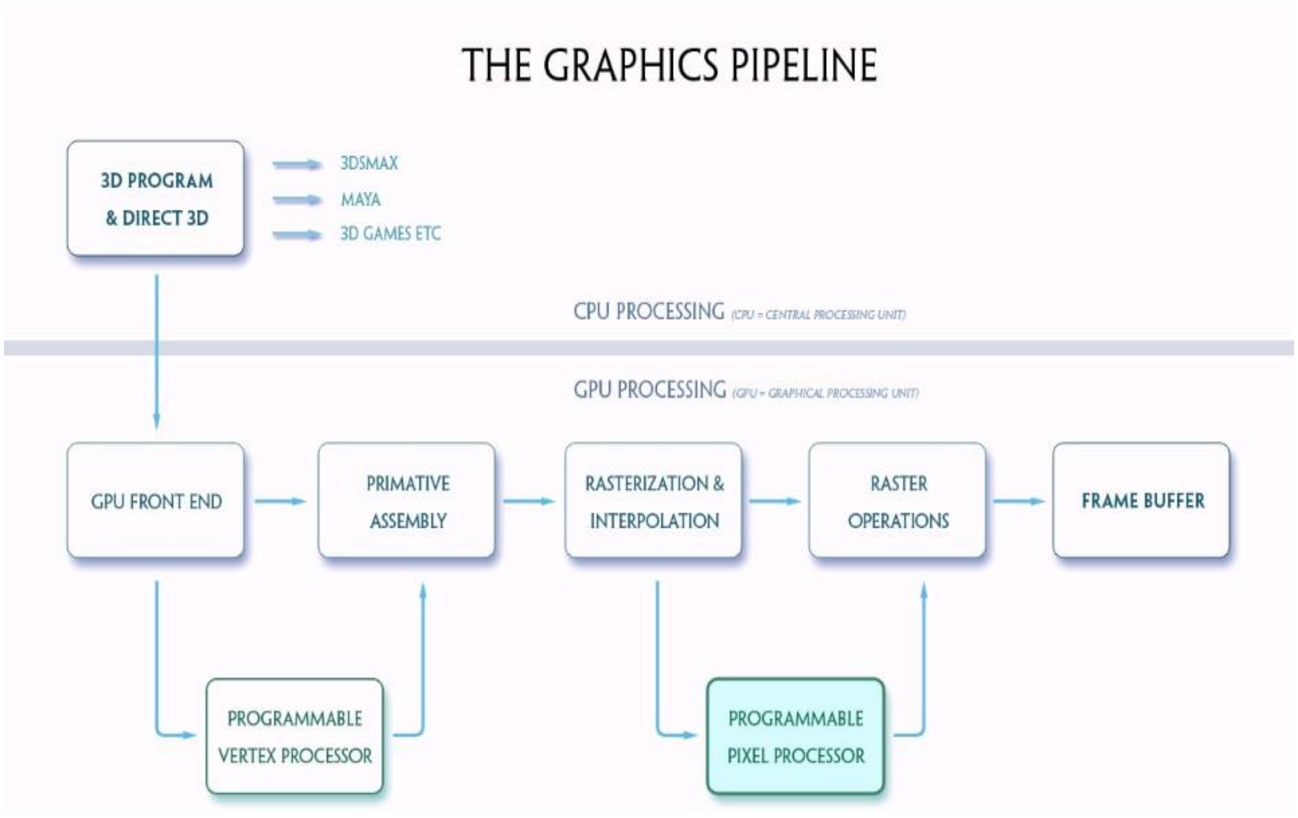
4.2 Types of Shaders and shading languages: -

- There are three types of shading languages available
  - HLSL(High level shading language) / Direct 3D
  - GLSL (Open GL shading language)
  - CG (by NVIDIA and Microsoft)
  -

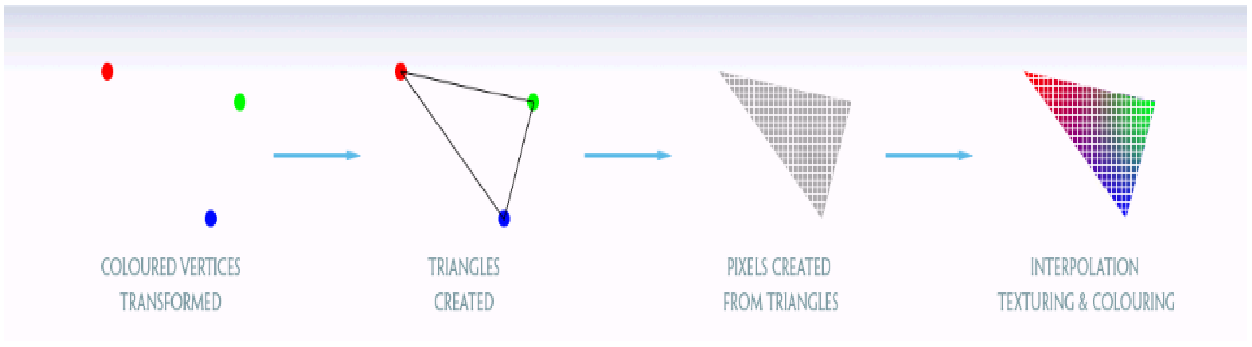
- Types of Shaders
  - **Vertex Shaders: -** Vertex shaders are run once for each vertex given to the graphics processor. The purpose is to transform each vertex's 3D position in virtual space to the 2D coordinate at which it appears on the screen (as well as a depth value for the Z-buffer). Vertex shaders can manipulate properties such as position, color, and texture coordinate, but cannot create new vertices. The output of the vertex shader goes to the next stage in the pipeline, which is either a geometry shader if present, or the pixel shader and rasterizer otherwise. Vertex shaders can enable powerful control over the details of position, movement, lighting, and color in any scene involving 3D models.
  - **Geometry shaders: -** Geometry shaders are a relatively new type of shader, introduced in Direct3D 10 and OpenGL 3.2; formerly available in OpenGL 2.0+ with the use of extensions. This type of shader can generate new graphics primitives, such as points, lines, and triangles, from those primitives that were sent to the beginning of the graphics pipeline. Geometry shader programs are executed after vertex shaders. They take as input a whole primitive, possibly with adjacency information. For example, when operating on triangles, the three vertices are the geometry shader's input. The shader can then emit zero or more primitives, which are rasterized and their fragments ultimately passed to a pixel shader.
  - **Fragments/pixel shaders: -** Pixel shaders, also known as fragment shaders, compute color and other attributes of each fragment. Pixel shaders range from always outputting the same color, to applying a lighting value, to doing bump mapping, shadows, specular highlights, translucency and other phenomena. They can alter the depth of the fragment (for Z-buffering), or output more than one color if multiple render targets are active. In 3D graphics, a pixel shader alone cannot produce very

complex effects, because it operates only on a single fragment, without knowledge of a scene's geometry.

4.3 Shaders GPU Pipeline and Data flow: -


THE GRAPHICS PIPELINE
3D PROGRAM & DIRECT 3D
3DSMAX
MAYA
3D GAMES ETC
CPU PROCESSING (CPU = CENTRAL PROCESSING UNIT)
GPU PROCESSING (GPU - GRAPHICAL PROCESSING UNIT)
GPU FRONT END
PRIMATIVE ASSEMBLY
RASTERIZATION & INTERPOLATION
RASTER OPERATIONS
FRAME BUFFER
PROGRAMMABLE VERTEX PROCESSOR
PROGRAMMABLE PIXEL PROCESSOR

CPUs normally have only one programmable processor.In contrast, GPUs have at least two programmable processors, the vertex processor and the fragment processor, plus other non-programmable hardware units. The processors, the non-programmable parts of the graphics hardware, and the application are all linked through data flows


COLOURED VERTICES TRANSFORMED
TRIANGLES CREATED
PIXELS CREATED FROM TRIANGLES
INTERPOLATION TEXTURING & COLOURING

The Cg language allows you to write programs for both the vertex processor and the fragment processor.We referto these programs as vertex programs and fragment programs,respectively. (Fragment programs are also known as pixel programs or pixel shaders, and we use these terms interchangeably in this document.) Cg code can be compiled into GPU assembly code, either on demand atrun time or beforehand. Cg makes it easy to combine a Cg fragment program with a handwritten vertex program, or even with the non-programmable OpenGL or DirectX vertex pipeline. Likewise, a Cg vertex program can be combined with a handwritten fragment program, or with the non-programmable OpenGL or DirectX fragment pipeline.

4.4 Simple Shader Program: -

- ***UI elements***

```
float4 AmbientColor : Ambient
<  string UIName = "Ambient Color";
= {0.25f, 0.25f, 0.25f, 1.0f};
float4 DiffuseColor : Diffuse
<  string UIName = "Diffuse Color";
> = {1.0f, 1.0f, 1.0f, 1.0f};
float4 SpecularColor : Specular
< string UIName = "Specular Color";
> = { 0.2f, 0.2f, 0.2f, 1.0f };
float Glossiness <
        string UIWidget = "slider";
   int UIMin = 1;
   int UIMax = 128;
   int UIStep = 1;
   int UIStepPower = 16;
```

```
    string UIName = "Glossiness";
          >= 40;
texture diffuseMap : DiffuseMap
<
    string name = "default_color.dds";
          string UIName = "Diffuse Texture";
    string TextureType = "2D";
>;
texture normalMap : NormalMap
<
    string name = "default_bump_normal.dds";
          string UIName = "Normal Map";
    string TextureType = "2D";
>;
```

- ***<u>Input from application</u>***

```
struct a2v
{
          float4 position : POSITION;
          float2 texCoord : TEXCOORD0;
          float3 tangent : TANGENT;
          float3 binormal : BINORMAL;
          float3 normal : NORMAL;
};
```

- ***<u>Output to fragment program</u>***

```
struct v2f
```

```
{
        float4 position    : POSITION;
        float2 texCoord  : TEXCOORD0;
        float3 eyeVec : TEXCOORD1;
        float3 lightVec : TEXCOORD2;
        float3 worldNormal : TEXCOORD3;
        float3 worldTangent : TEXCOORD4;
        float3 worldBinormal : TEXCOORD5;
};
```

- ***<u>Vertex shader part …</u>***

```
v2f vertex(a2v In, uniform float4 lightPosition )
{
        v2f Out;
        - - - - - - - - - - - - - - - -
        - - - - - - - - - - - - - - - -
        //vertex shader program goes     //here
        - - - - - - - - - - - - - - - -
        - - - - - - - - - - - - - - - -
        return Out;
}
```

- ***<u>Fragment / pixel shader part …</u>***

```
float4 fragment(v2f In,uniform float4 lightColor) : COLOR
{       float4  color_val ;
        - - - - - - - - - - - - - - - - - - - -
        - - - - - - - - - - - - - - - - - - - -
        //pixel shader program goes       //here
        - - - - - - - - - - - - - - - - - - - -
        - - - - - - - - - - - - - - - - - - - -
        return color_val;
}
```

Techniques and passes…

- Techniques are the variations in the shaders
- Passes are one complete trip down the graphics pipeline

```
technique regular
{   pass one
   {    VertexShader = compile vs_1_1 v(light1Pos);
        ZEnable = true;
        ZWriteEnable = true;
        CullMode = CW;
        AlphaBlendEnable = false;
        PixelShader = compile ps_2_0 f(light1Color);
   }
}
```

5. ***References***

- www.ogre3d.org
- www.crydev.net
- Nvidia official web page
- Nvidia cg tool kit users manual
- Wikipedia.