

3D Engines in games - Introduction

Author: Michal Valient (valient@dimension3.sk)

Conversion (with some corrections) from HTML article written in April 2001.

Foreword and definitions

Foreword

Real-time rendering is one of the most dynamic areas in the computer graphics (later only CG). Three dimensional computer games are in other way one of the most profitable commercial applications of the real-time rendering (and CG as whole). Real-time rendering attracts more and more people every year. With every generation of 3D accelerators we see nicer and more realistic games with new effects and more complex models. This article is meant to be a small introduction to the field of real-time graphics for computer games.

The article is divided into several sections

- History of 3D games – a brief history of real-time 3D games, accelerators and API's.
- Game engine scheme - parts of generic game engine with description.
- 3D API basics - very basic description of the pipelines.
- World of games - describes the specific world of games, point of view of the gamer and point of view of the programmer.

Definitions and identifications

- **Real-Time Rendering** - means a process, when on the screen is displayed a picture, user makes a response and this feedback has an effect to what is rendered during the next frame. This cycle is fast enough to fool user, that he doesn't see individual pictures, but smooth animation. Speed of rendering is measured with fps (Frames Per Second). One fps is not an interactive process. With six fps feeling of interactivity rises. Fifteen fps allows user to concentrate to action and reaction. Upper limit is 72 fps because of limitation of an eye (more in [1]).
- **3D** - denotation for three-dimensional space.
- **Pixel** - screen point (from Picture Element).
- **Vertex** - denotes point in space with its other properties like normal, color, transparency and others. For example a triangle (or **face**) is defined in 3D with three vertexes and additional data (normal, texture ...).
- **Texturing** - process which takes a surface and modifies its appearance in every area with some picture, function or other source of data. This source of data, picture or function is denoted as texture. One texture point is named **texel**.
- **Billboarding** - method of replacing potentially complex (3D) object with it's 2D representation (**a sprite**) rendered from some point of view and showing this sprite upright to the camera no matter how camera is rotated. More in [1] on page 152.
- **Mipmapping** - process of choosing a texture from pool of (identical) textures with various resolutions according to the distance of textured object. The smallest texture is used on the farthest object and high resolution texture is used on near object.

- **Lightmapping** – a method of shading. Objects are shaded using for example radiosity or raytracing and shade for every triangle (or whatever is used) is stored in the texture. Material of the triangle is modulated with this texture during rendering to improve the realism of the scene. An advantage is that this method is fast at render time. Disadvantage is that this method is computationally very intensive and therefore cannot be used for moving lights or objects.

History

History of 3D real-time rendering in the computer games falls back to year 1984 to the 8-bit computer ZX Spectrum (i.e. game Zig-Zag, sorry no picture). On this computer with 4 colors and memory of tens of kilobytes was 3D game a peak of art. Later more powerful 8-bit computers came like the Amiga (also 16-bit version) and the Atari. The graphics (mostly on Amiga) was much better.

It was a revolution in the world of entertainment when first 16-bit PCs came and company id Software (and mostly the programmer John Carmack) created the game Wolfenstein 3D back in 1992. It was the first game where textures were used. All objects were painted using the billboard method. It's true, that there was only one rotation axis computed (left/right), only (perfectly vertical) walls were painted with the texture and all levels were basically 2D but popularity of that game (and sequel Spear of Destiny) indicated what people want to play. (Update 2004 - In the same year Ultima Underworld was released and featured fully textured 3D engine)



Pictures from Wolfenstein 3D and Spear of Destiny

In the year 1993 id Software overcame themselves and the game DOOM advanced the limit of what is possible. Game works in higher resolutions, textures are used also on the roof and the floor and the rooms are no more flat, but full of steps and rises. Rotation was possible around two axes (left/right and up/down) and all moving objects were rendered using billboards. The game introduced network deathmatch and cooperative mode. John Carmack and id Software are legends from that moment. Game had a sequel named DOOM2.



DOOM1 and DOOM2

Next revolution game was released in the year 1995 - Descent by Parallax software. It allows movement and rotation in every direction. Attempts with the Virtual Reality helmets began in that time, but because of low resolution of the head mounted displays and small computational power of the PC they were not successful.



Descent

The Quake was released in the year 1996 (again Carmack and id Software). And it was a breakthrough in the 3D computer games. Every in-game model was fully 3D. Other features are - mipmapping, dynamic scene lighting (vertex only) and static objects were shaded using lightmaps.



Quake



Quake

The Year 1997 was revolutionary in 3D games. First 3D accelerator for the home PC was created and it was Voodoo from 3Dfx. With hardware implemented features like bilinear filtering of textures, fog and perspective correct texture mapping allowed the developers to skip writing the hardest part of engine code - low level routines for painting textured triangles fast and visually correct. Games were nicer (due to filtering), faster and with more details. It was only a question of time, when accelerators would be mandatory for 3D games. OpenGL, in that time the domain of professional SGI workstations, started to be used widely by game developers. It is a non object oriented library that allows quality and fast rendering both in 2D and 3D. OpenGL specification is defined and upgraded by the Architecture review board – ARB - (community of companies like SGI, NVIDIA, 3D Labs and others), so it is an open standard. However this approach has some disadvantages. OpenGL specification reacts very slowly to the new trends (pixel and vertex shaders are not in the version 1.3 [Update 2004 – we have updated OpenGL 2.0 now with shaders]). The system of extensions (functionality not directly available in OpenGL can be plugged in via an extension) is not controlled and far from perfect because every hardware manufacturer defines own extensions that works only with specific accelerators (NVIDIA and ATI are perfect examples). Developers have to write specific code to some combination of available extensions and in fact for concrete accelerator. This is the opposite to the meaning of the OpenGL - to be hardware independent API. First top selling game using the accelerator and OpenGL was Quake II (of course by id Software)



Quake 2



Quake 2

Microsoft started development of the DirectX - an object oriented library that allow very low access to graphics sound and input hardware from Windows. Graphics part of the library is a rival of OpenGL. DirectX is developed in the cooperation with major hardware manufacturers and this allows faster reaction to the new trends (probably because ARB is not the primary interest of its members and therefore changes come later than in DirectX).

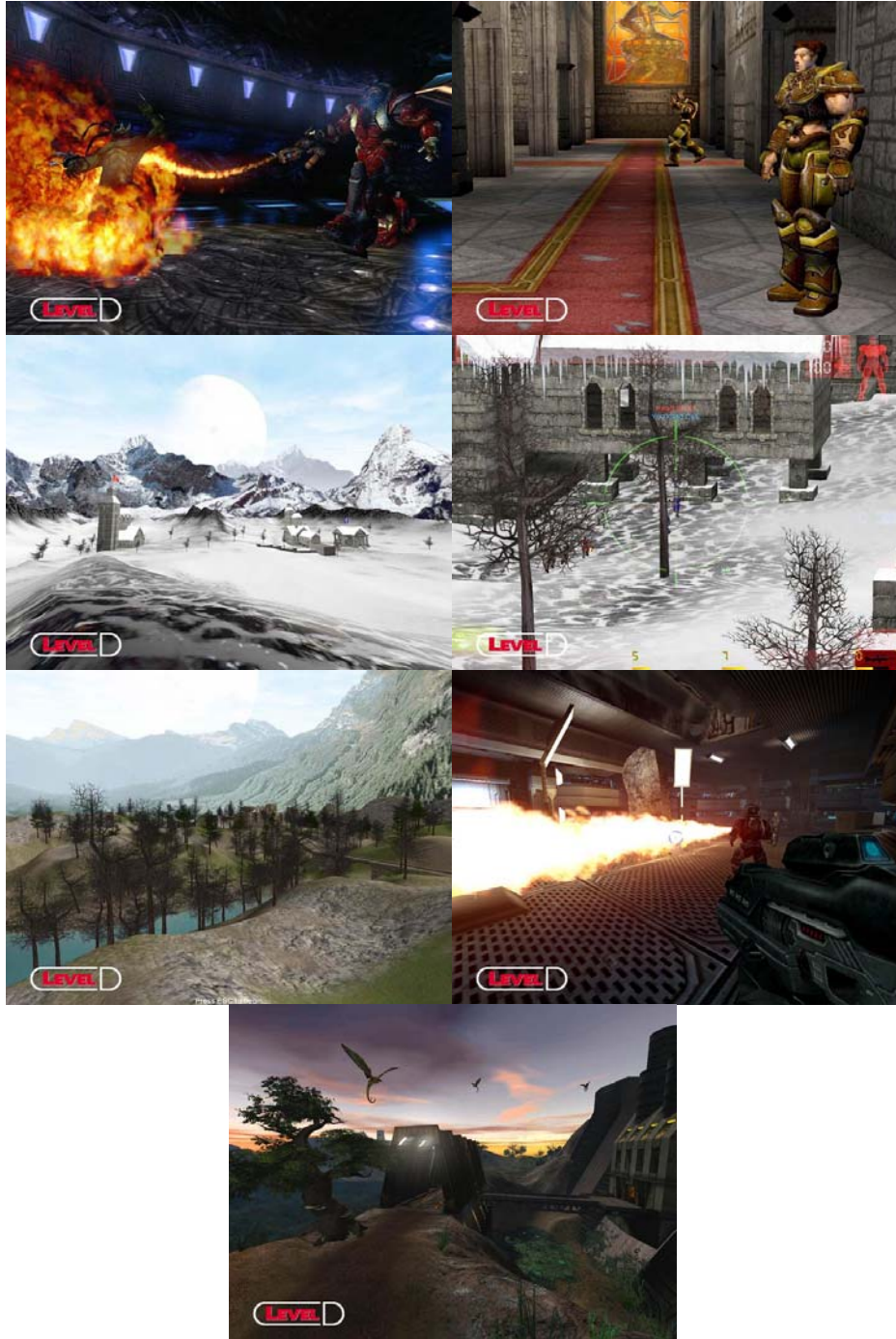
Accelerators become faster and several minor features were added (i.e. multi-texturing, anisotropic filtering) but it was more evolution than a revolution. New graphics chip GeForce256 was released in the year 1999. The author, NVIDIA, called it GPU (geometry processor unit) because it was not simple renderer of triangles (Voodoo is the example where transformed and lit triangle came in, texture was applied and it was drawn). GeForce256 offers transformations and lighting (T&L) implemented in hardware and computes them instead of CPU. T&L was implemented into DirectX from version 7.

Another revolutionary change to the architecture was introduced in the year 2001. NVIDIA released the GeForce3. This chip goes beyond T&L and offers programmable vertex and pixel processing pipeline. This allows developers to create custom transformations and effects that were not possible with fixed T&L pipeline. ATI has developed similar (Update 2004 – and better) chip called Radeon 8500. DirectX 8 adopted this feature in the system of vertex and pixel shaders. Shaders in the OpenGL are available only via company specific extensions (Update 2004 – we have ARB extensions now and OpenGL 2 does have shaders). The game console Xbox by Microsoft contains modified Geforce3 chip and therefore number of games using shaders are currently in the development.

In the end of year 2002 we expect new version of DirectX API and new wave of accelerators to be released. They will offer increased programmability and speed.



Quake 3 and Return to castle Wolfenstein uses Quake 3 Engine and has limited support for shaders on Geforce 3



Unreal2 engine



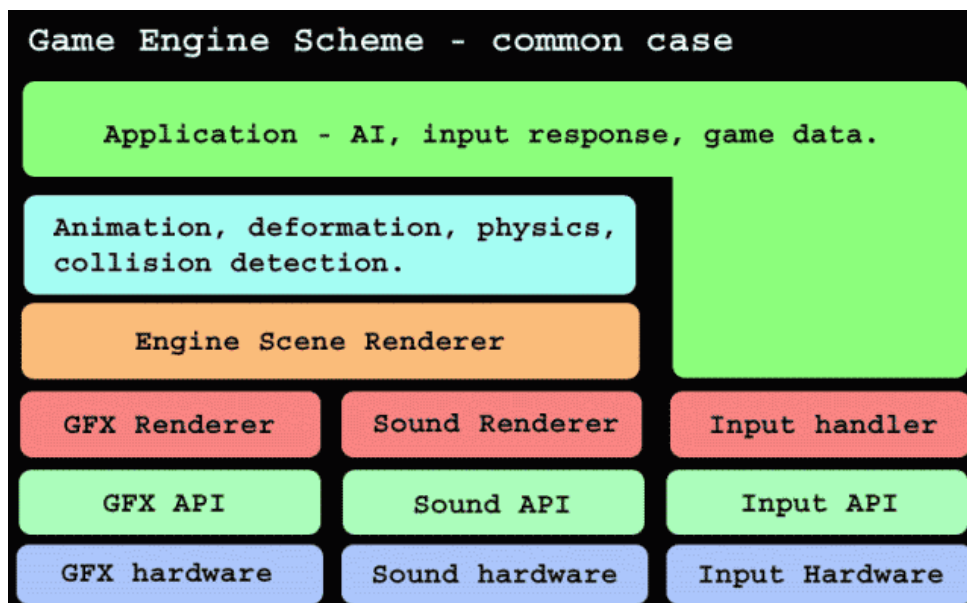
Comanche 4 – a game that uses shaders

The history of games is also interesting from the business view. In the beginnings only a few people were creating games. Later small teams, created by a few friends, created spectacular games. Now we have huge game industry with earnings comparable to the movie industry that uses Internet hype and loyal people as the propagation channel.

3D Engine

The scheme of 3D general engine

First let us define what the Game Engine means in this document. Game Engine (or 3D Engine) is a complex system, responsible for the visualization and sound of game that handles the user input and provides resource management, animation, physics and more.



On the lowest level there is hardware. API that accesses hardware is one level above. In MS Windows it could be DirectGraphics (formerly Direct3D) or OpenGL for visualization. For sound it is DirectSound, OpenAL or some of the other available libraries, input can be handled via DirectInput. GFX Renderrer is responsible for drawing of the final scene with all the fancy effects. Sound renderer has similar position in the audio field and plays sounds from correct positions using correct effects. Input handler gathers events from keyboard, mouse or other input devices and converts them to format acceptable by the engine. It is also responsible for correct commands to the force-feedback devices. Engine scene renderer uses lower level methods to render the scene to the screen and play correct sounds. The layer above renderer has a lot of functions, which cooperate in-between. It includes animation (timeline work, morphing and manipulation with objects depending on time), collision detection (it influences animation and deformations). Deformation uses physics to modify shapes depending on forces. Physics computes gravitation, wind and more. This layer might have a lot of other functions, but they are application (game genre) depended. Above this is the application. It is responsible for game data, AI (but this might be in the lower layer), GUI, response to the user input and all the other stuff that

makes us staring at the screen and saying “whoa...”

This structure is based on my studies of a lot of various engines and for sure it is not complete, not deep and maybe time shows that it is even not good. All comments are welcome.

API Basics

Direct3D and OpenGL rendering basics

“What graphics API would I use” - is one of the first decisions the developer has to make when starting creation of a new 3D Engine. There are three possibilities today: OpenGL, Direct3D or create an API independent engine that can use both of them. To make this decision a little bit easier take a look at following two lines:

```
glDrawElements(X,Y,Z);
```

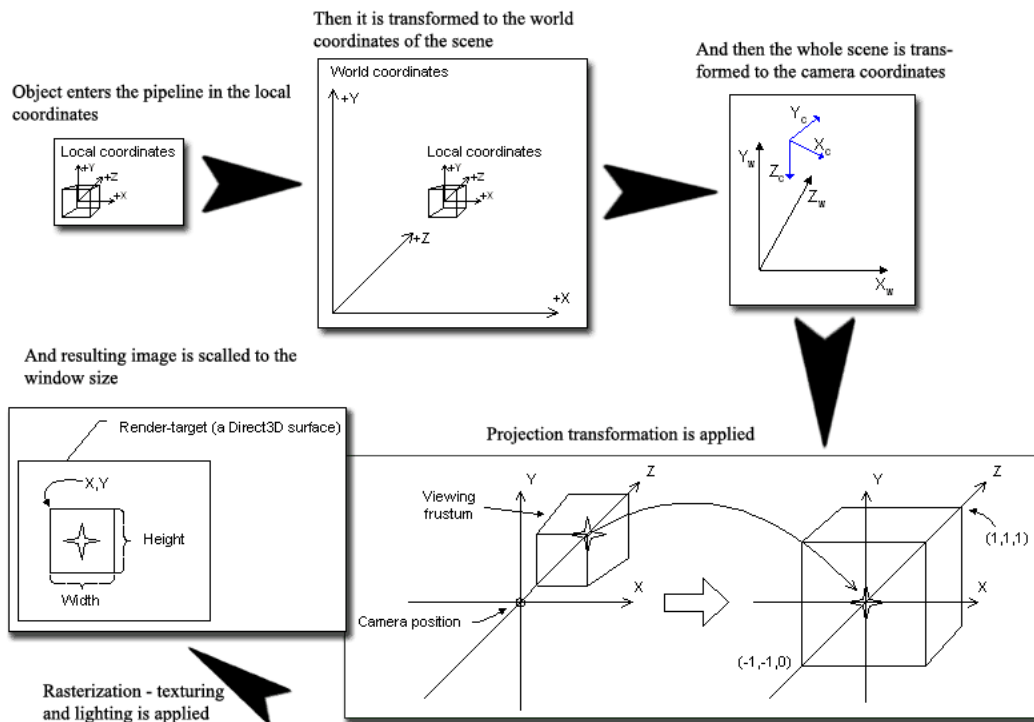
```
direct3DDevice->DrawIndexedPrimitive(X, Y, Z);
```

A programmer, who likes object oriented approach and does not plan to compile engine under OS other than Windows (or maybe Xbox) could choose DirectX. The others could use the OpenGL. Each of these alternatives has the pros and cons. DirectX is upgraded very dynamically and it already contains support for input, sound and network. It is supported on Windows and Xbox. On the other hand DirectX does not guarantee you that feature not supported by hardware is supported by software and provides a system of capabilities flags so developer can easily detect what can and cannot be done (in games, where we are fighting for highest fps is software nevertheless out). OpenGL from this point of view was mentioned on previous pages.

Rendering with accelerators has besides a lot of advantages (speed, quality, easier implementation) also some drawbacks. They all have common root: we have to adapt to the hardware we are using. We cannot choose our private format for textures, vertices or lights. We cannot modify data submitted to accelerator at any time. The others cause very poor performance: if we want speed, textures and meshes have to be uploaded to the private memory of the accelerator and let it choose the best format. By doing this we cannot modify these resources without costly lock operation that downloads the data back to system memory. This can be solved by double buffering of the data. One set for us, one set for the accelerator. Changing active textures very frequently is also a quite time consuming operation. This can be solved (not absolutely) by rendering primitives with identical materials in one batch.

Inside the API - Fixed function pipeline

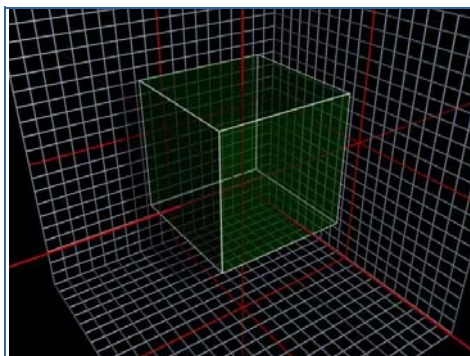
Fixed Function Pipeline



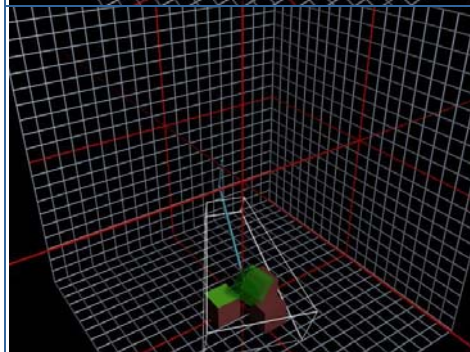
Fixed function pipeline (a compilation of images from DirectX SDK)

Now it is a good time to show what is done inside the API when we want to show a cube using the fixed function pipeline (supported both in DirectX and OpenGL).

The geometry and lighting phase:



On the input we have our test subject (a cube this time) in the local coordinates - vertices are positioned relative to a point that represents the center of an object. It is a good practice to define the point as a most used center of rotation and scale of the object.



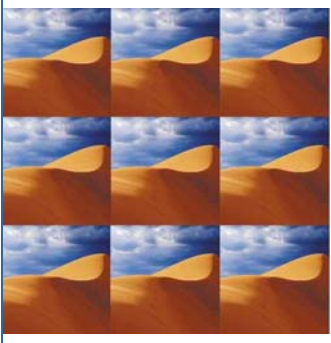
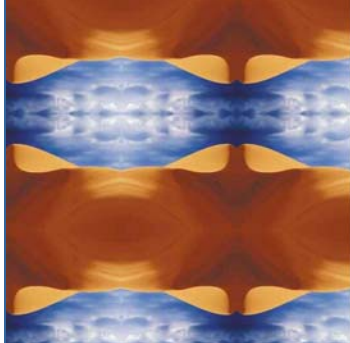
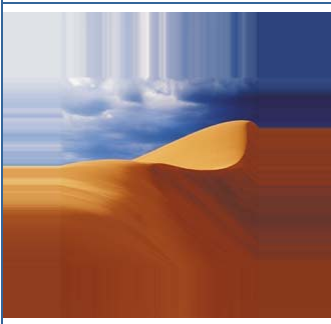

Our cube goes to the next stage with a matrix that will transform it to the world coordinates - in most cases we don't want every object to be placed on $[0, 0, 0]$ but somewhere else and rotated and scaled somehow.

	<p>The next step in the pipeline is a transformation of the whole scene to the camera space. If we have camera in the point $[2,2,2]$ we transform whole scene so that camera is in $[0,0,0]$ and the up vector will be identical to the y axis.</p>
	<p>Now is time to (likely) first transformation that changes the shape of our cube. This is because if we use perspective projection, API has to transform it to parallel.</p>
	<p>If we have the backface culling enabled, invisible faces are removed and if we have the lighting enabled, scene is lit. Texture coordinates are applied depending on the specified texture transformation.</p>
	<p>Results of the scene are transformed to the viewport coordinates of a window and let's go to rasterization.</p>


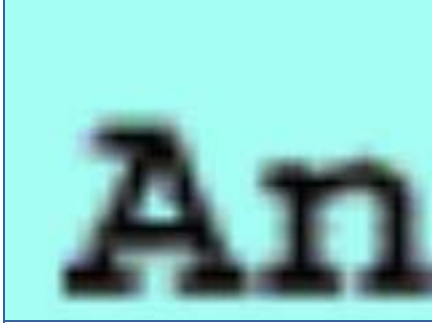


The rasterization **phase:**

Our model passed through whole geometry stage process and now it is being painted to the output. How this is done depends entirely on the operation used during rendering (replace, add, multiply, subtract, and, or). Every face can have up to 8 textures (in DirectX8). The way how these textures cooperate and look is defined by the address mode, the filter mode and the texture blend function.

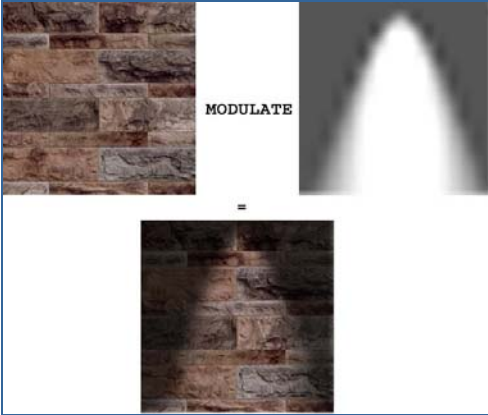
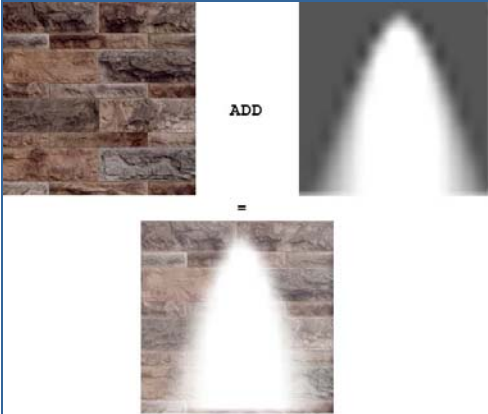
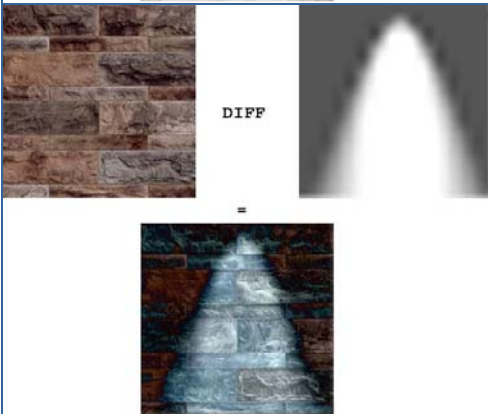
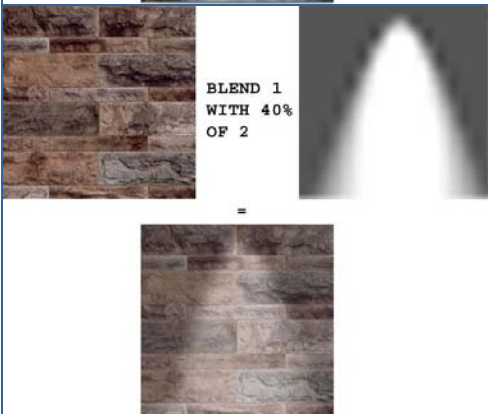
The texture addressing modes:

	<p>Wrap. Texture is tiled. If we use face coordinates [0,0] [0,3] [3,3] texture is repeated 3 times in every direction.</p>		<p>Mirror. Very similar to wrap, but texture is mirrored before the repeat.</p>
	<p>Clamp. Coordinates outside the range 0 and 1 are clamped to the nearest border pixel of texture.</p>		<p>Border. Coordinates outside the range 0...1 specify just a single predefined color.</p>

Texture filtration modes:

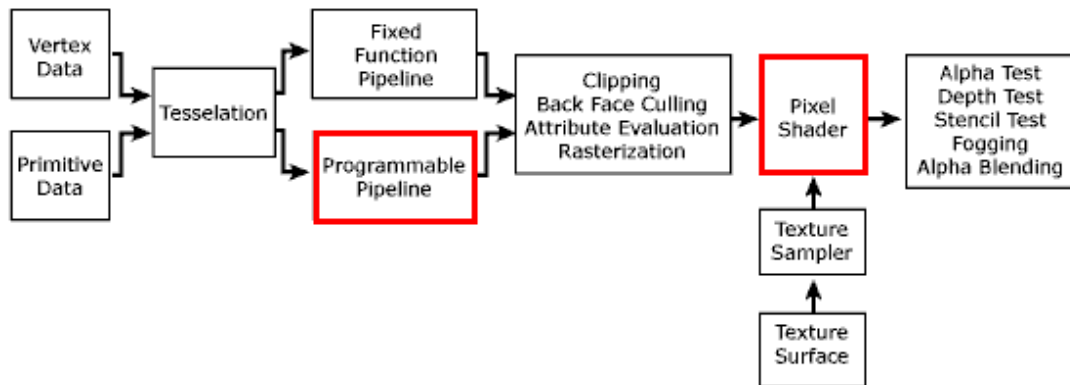
	<p>Point - nearest defined neighbor is used when finding texel color</p>
	<p>Bilinear - bilinear interpolation of 4 neighbors is used when finding texel color.</p>
	<p>Trilinear - linear interpolation of two nearest bilinearly filtered mipmaps is used.</p>
	<p>others - anisotropic, flat cubic, Gaussian cubic</p>

Texture blend functions (texture 1 is blended with texture 2, output is blended with texture 3 and so on):

 <p>The diagram shows a stone texture (source 1) and a grayscale gradient (source 2) being multiplied together. The result is a stone texture where the brightness is modulated by the grayscale gradient.</p>	<p>Modulate - multiply RGBA parts of source 1 with RGBA parts of source 2</p>
 <p>The diagram shows a stone texture (source 1) and a grayscale gradient (source 2) being added together. The result is a stone texture with a bright white glow from the grayscale gradient.</p>	<p>Add - add RGBA parts of source 1 to the RGBA parts of source 2</p>
 <p>The diagram shows a stone texture (source 1) and a grayscale gradient (source 2) being subtracted. The result is a stone texture where the grayscale gradient is inverted and applied, creating a dark, shadowed effect.</p>	<p>Subtract - Subtract RGBA parts of source 1 from RGBA parts of source 2</p>
 <p>The diagram shows a stone texture (source 1) and a grayscale gradient (source 2) being blended. The result is a stone texture where the grayscale gradient is applied with 40% opacity.</p>	<p>Blend - linear interpolation between RGBA parts of source 1 and RGBA parts of source 2 controlled by parameter.</p>
<p>Other:</p>	<p>disable, select, dotproduct3.</p>

Inside the API - Programmable pipeline

DirectX 8 introduced programmable geometry and pixel pipeline. In the OpenGL it is available via extensions on the NVIDIA and ATI hardware.



Programmable pipeline

Geometry and lighting phase is now completely replaced by an assembly like program loaded and compiled at runtime. This means developer can do anything he wants with vertices. The rule is: one vertex comes in, one comes out. Communication between vertices is not possible (due to the high level of parallelism in the hardware).

Rasterization phase was replaced too and texture blend functions are replaced by pixel shader language.

For more information about shaders visit OpenGL.org or MSDN DirectX page.

World of games

There are two points of view that influence the game development. The point of view of a gamer that plays the game and the point of view of a developer that has to create a successful game.

View of gamer:

- **Speed**. Gamers want smooth movements at very high frame rates. You can create one very cool looking game, but if it is slow on an average machine, people will drop details and in the result it will look like *Wolfenstein3D* because game was not running at 70 fps (i.e. *Quake 3*, *Counterstrike* and other first person shooters [2004 – or *Doom3* and *Far Cry*]).
- **Reality**. Gamers want to play as real as possible, but not too much.
 - If you create a hyper-realistic space simulator that uses every single key on the keyboard to control the ship, you'll fail (in hyper-realistic space there is no sound and that's boring and player is not an octopus). If you create something that is controlled by joystick, ten keys and ships moves a little bit like in the space (or like the player thinks ship has to move after the *Armageddon* movie) and if you don't drown it with awful graphics, sound and marketing you could win and be rich.
 - If you create an ultra-realistic car chase then majority of players will be most of the time out of track and game could be successful only in the community of hardcore F1 fans (most probably only Schumacher

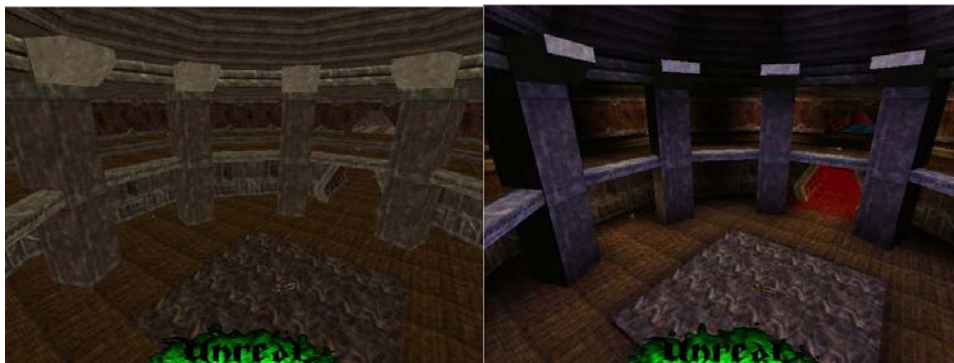
brothers). But if it's easy to drive, skids are showy and easy to handle and crashes look like scenes from a blockbuster movie, then it can be successful game.

Players do not want to die after every grenade blast in the first person shooters that do not look like Second World War simulation.

- **Effects.** Game must be good looking. A lot of lens flare effects (you know they are not visible if you are not looking through camera, but who cares if they look good). A lot of smoke, dust and fire (we love it). And let's have nice specular highlights and shadows.

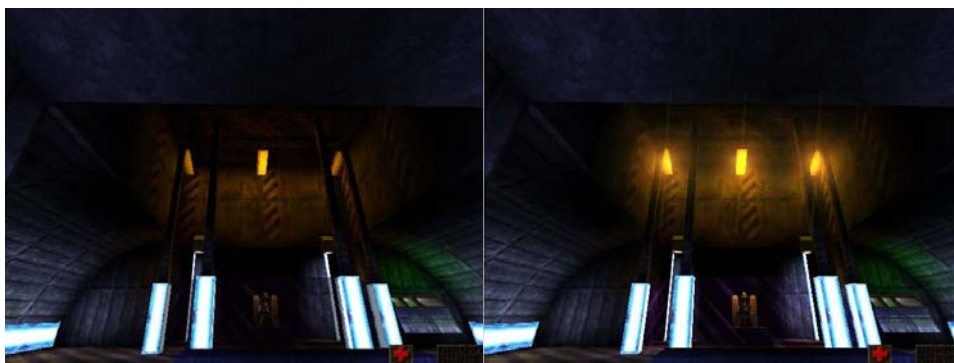
The point of view of a programmer (or how to make all that was mentioned above running in real-time):

- **Shadows** can be pre-computed and then used as the lightmaps or they can be computed in real-time for example using projective shadows, shadow volumes or shadow mapping.



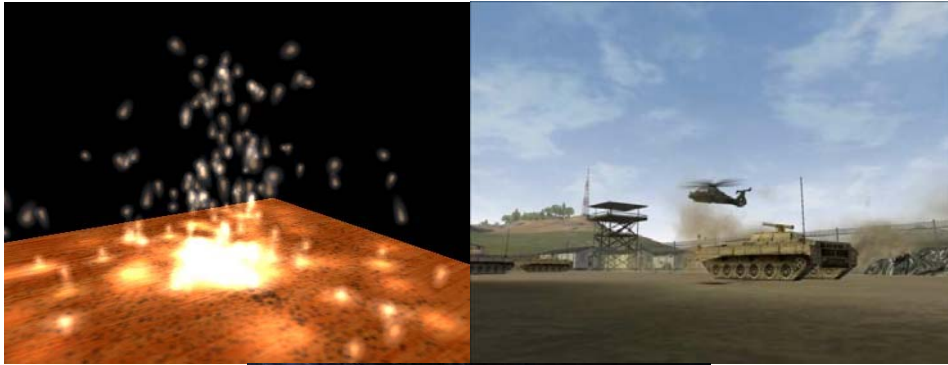
No shadows versus lightmap shadows (images from game Unreal by Epic)

- **Atmospheric effects** like lens flares and halos – some parts of these effects are stored as textures and then displayed after whole scene is rendered using ADD method. Using them provides if not more realistic, then at least more atmospheric feel from game.



Screen without and with halo and lens flare effects (images from Unreal by Epic)

- **Fire, dust and smoke.** Today's most popular method is using the particle systems (high count of simple squares with texture and they are displayed always upright to camera and blended to the scene). Color and velocity of a single square depends on its lifetime, distance from emitter and distance from other squares).

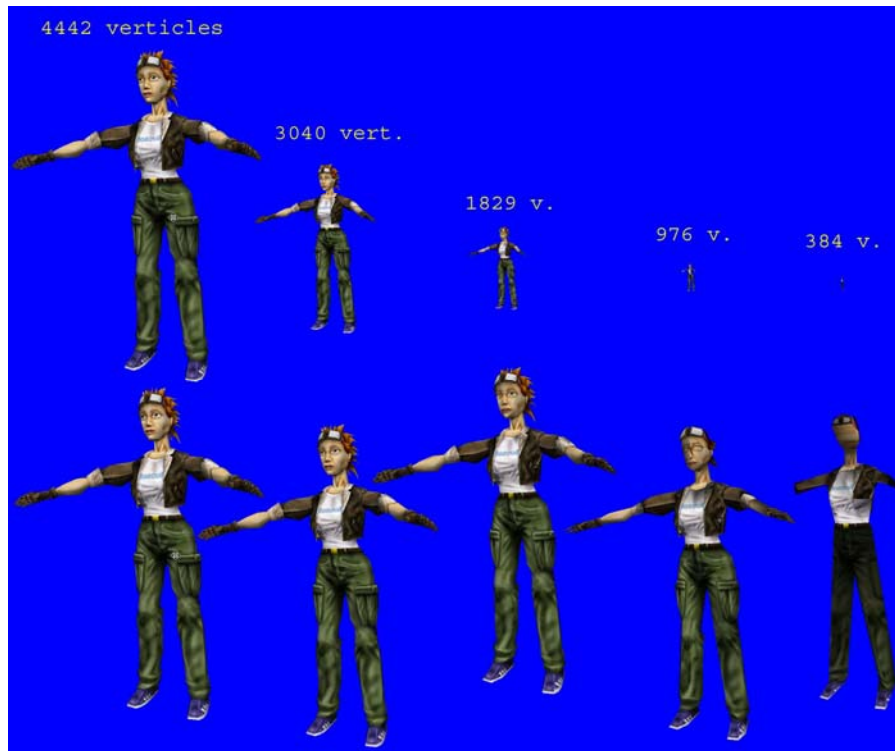


Particle simulation of sparks, dust and flame (images from nVidia SDK sample, Comanche 4 and Unreal 2)

- **Objects.** Players want high detail objects, but if every object in game would have ten thousands of faces it will not be possible to draw the scene in real-time rates. Therefore several methods of eye-fooling are used. Trees and plants are billboarded. Objects that are far from the camera (or that are small on the screen) are rendered using reduced triangle count (method called LOD - level of detail). This method is similar to the mipmapping except that instead of multiple resolutions of textures we use multiple instances of the same object with different triangle count.



Billboarding (image from nvEffectBrowser by nVidia)



LOD (image from DirectX 8.1 SDK sample)

- **Animations**. When animating a living creature it is not very good looking if the leg is divided from the body like it is a robot. So we need to use the creature as single model. Animation is then provided using mesh morphing (we have finite set of object states and we perform interpolation between different states) or mesh skinning (we have model and a set of bones for this model. Moving bones causes model to animate).



Mesh skinning image from nvEffectBrowser by nVidia)

- **Reflections**. On the perfectly flat surfaces (a floor or a mirror) it can be computed in real-time (scene is rendered twice for every mirror and if two mirrors can see each other we have a problem to solve). On the curved surfaces we use reflection maps. Reflection maps are textures that are mapped onto the object depending on the camera position. Single texture (that specifies a hemisphere), two textures (specify a paraboloid) or cube maps (6 textures) can be used. Advantages are that it can be updated in real-time and can have smaller resolution depending on the object size.

Environmental mapping - texture holds scene reflection from object painted to hemisphere that is visible by camera. Hard to update real-time, but can be prepared offline.

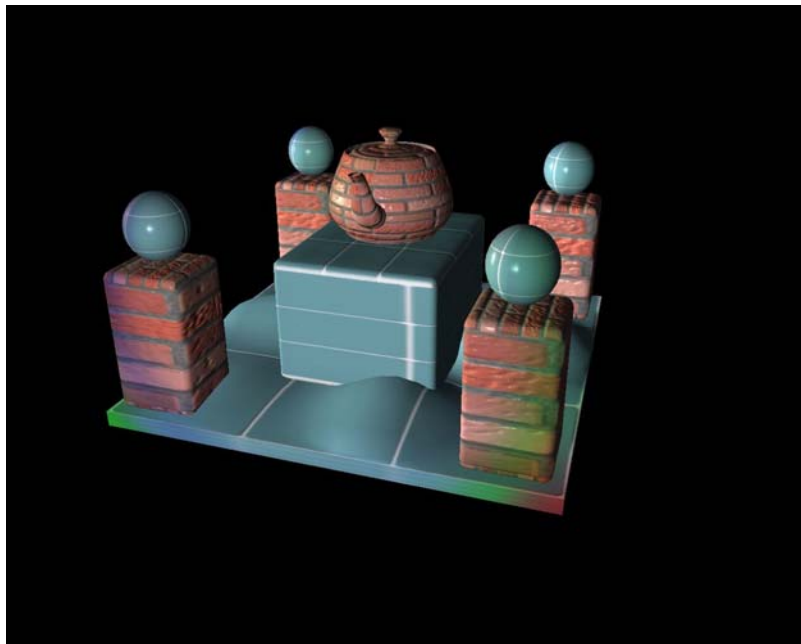
Paraboloid - two textures with back and front hemisphere. Hard to update real-time, but can be prepared offline.

Cube mapping - 6 textures are used as sides of cube that is bounding rendered object. Each contains screen as "seen from object" in that direction. Can be computed at real-time, is easier to use in rendering but has bigger memory requirements.



Cube mapping example (image from DirectX 8.1 SDK)

- **Lighting.** For gamers it is equal if game uses per-vertex or per-pixel lighting as long as they look the same. We can use different types of lighting on different objects to achieve decent framerates.



Per pixel lit cube with gloss map

- **Scene partitioning** - rendering the whole scene with all objects that are not visible is a bad idea and with more complex scenes it can result in very low framerates. Because of this portals and tree structures can be used.
 - Portal - invisible surface (to a gamer, not to the engine) that divides two (usually convex) cells of scene. If we are rendering one cell and the portal is visible to the engine, then the other cell is considered as visible and we need to render it too. This is used mainly for the in indoor scenes.
 - Trees - we can group objects into the cells and then divide these cells into

sub cells. If parent cell is not visible, then the child nodes cannot be visible too. This is used for the outdoor scenes or very large visible and dynamic areas.

Literature

- [1] Möller T., Haines E., "Real - Time Rendering", A.K Peters, Natick, Massachusetts, 1999, <http://www.acm.org/tog/resources/rtr/> or <http://www.realtimerendering.com>

Other resources

- www.idsoftware.com - all information about id games
- www.gamespot.com - pictures
- msdn.microsoft.com- DirectX reference.
- www.opengl.org - OpenGL reference.
- www.level.cz - pictures.
- www.score.cz - pictures.