3D Computer Graphics

A "Solid" Introduction

Lorenzo Linari - 02/11/2016

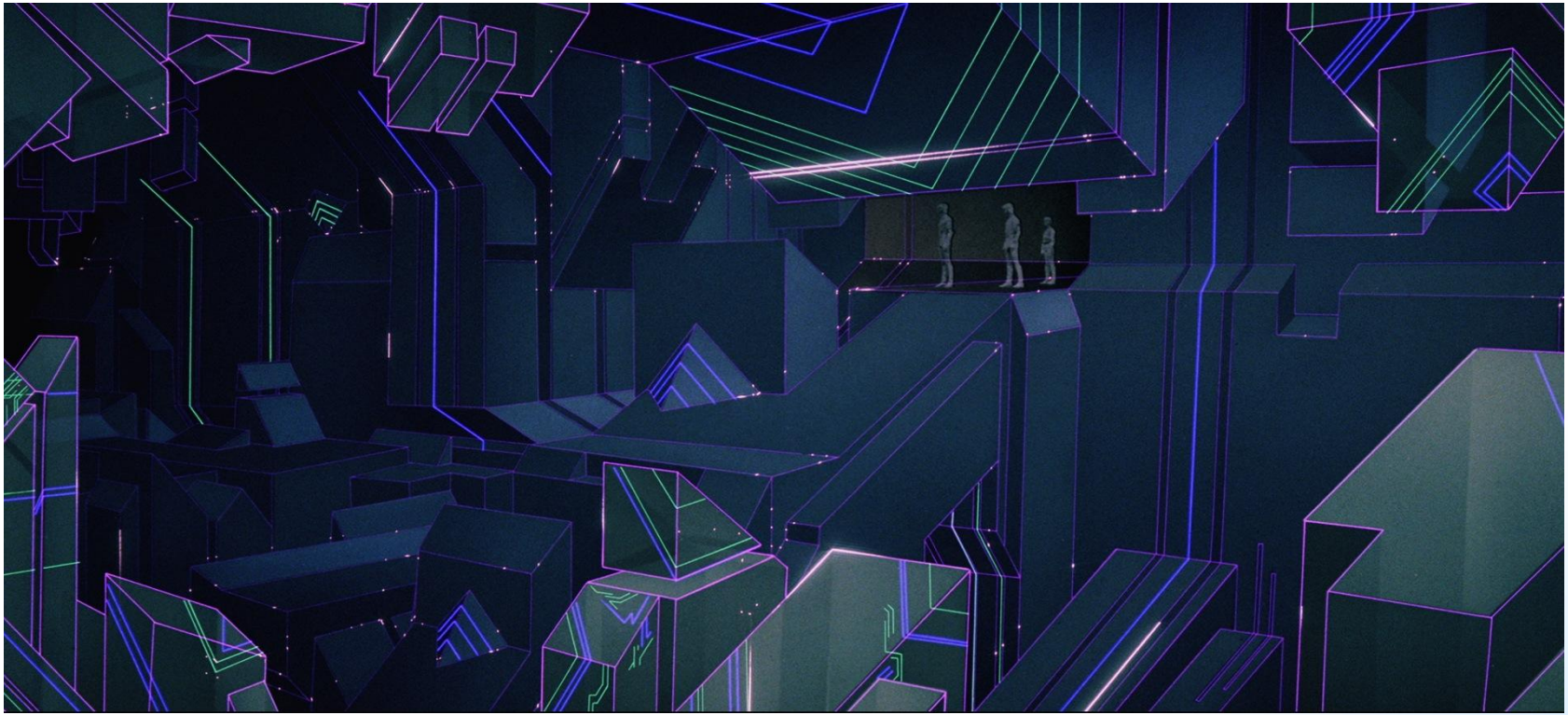
What is 3D Computer Graphics

"3D computer graphics (in contrast to 2D 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. Such images may be stored for viewing later or displayed in real-time."

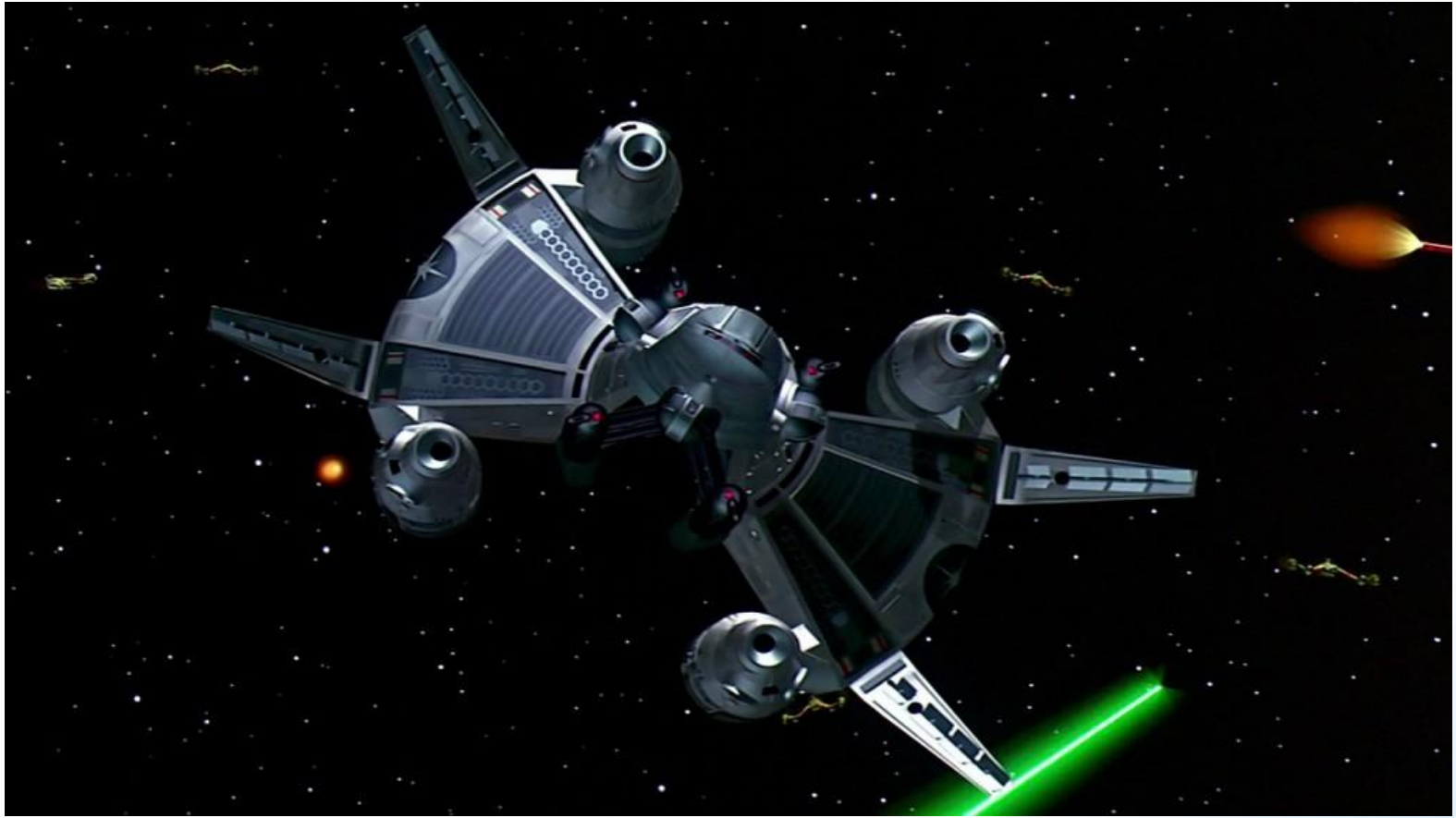
Wikipedia

Offline Rendering

- Offline Rendering aim is to produce the best possible visual result without time computation constraints.
- Algorithms for photorealistic rendering are quite expensive due to the complex calculations needed to reproduce real-world physical effects like lighting, materials, reflections, translucency …
- Once rendered, the result cannot be modified without a full recomputation.
- Commonly used in Movies, Photorealistic Renderings for Industrial and Architectural purposes. Good enough even in the 80's.
- Thanks to GPUs advancement, historical offline algorithms like radiosity are now possible also in real-time.



Tron (1982)



The Last Starfighter (1984)

Star Trek (2009)

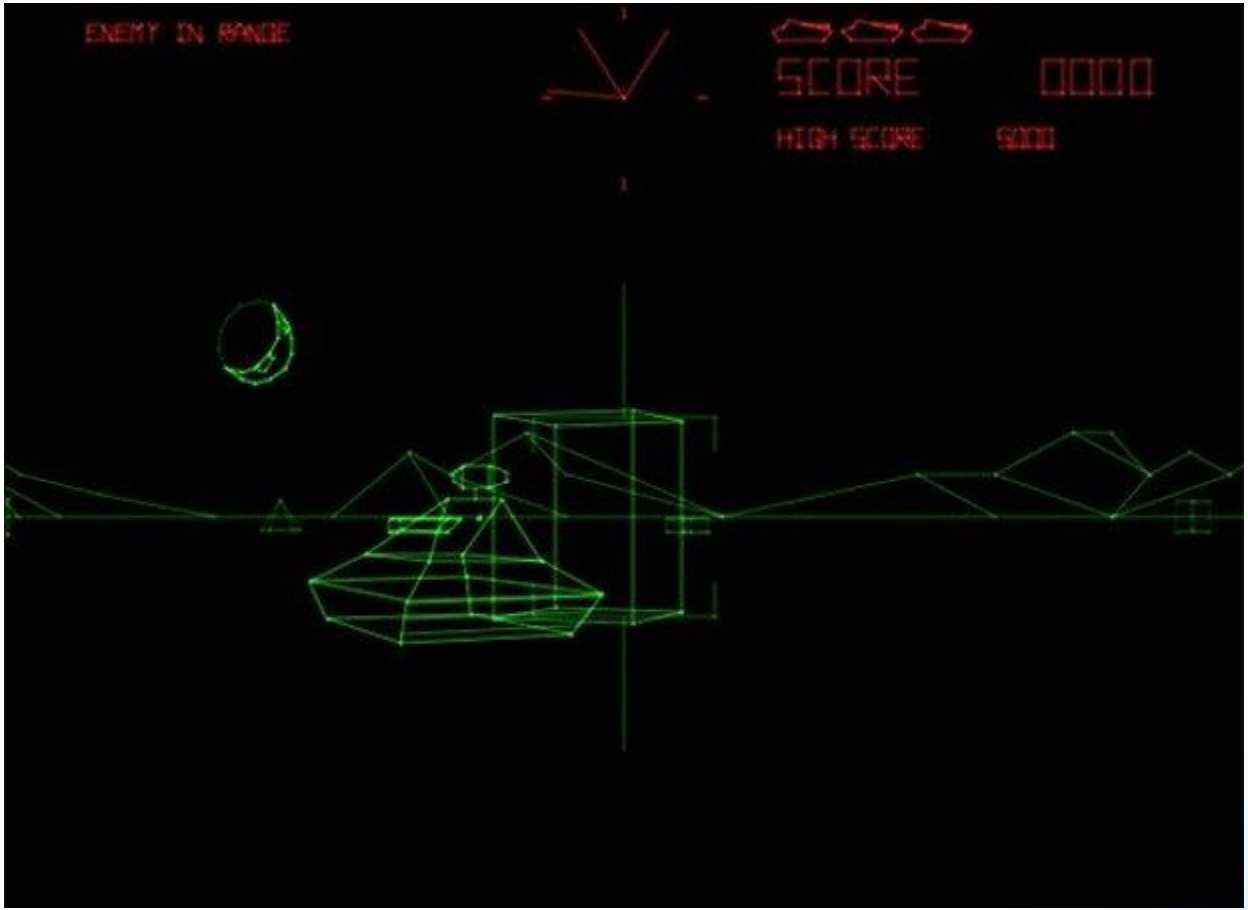
Octane RayTracer

my3dview.com.au

Octane RayTracer

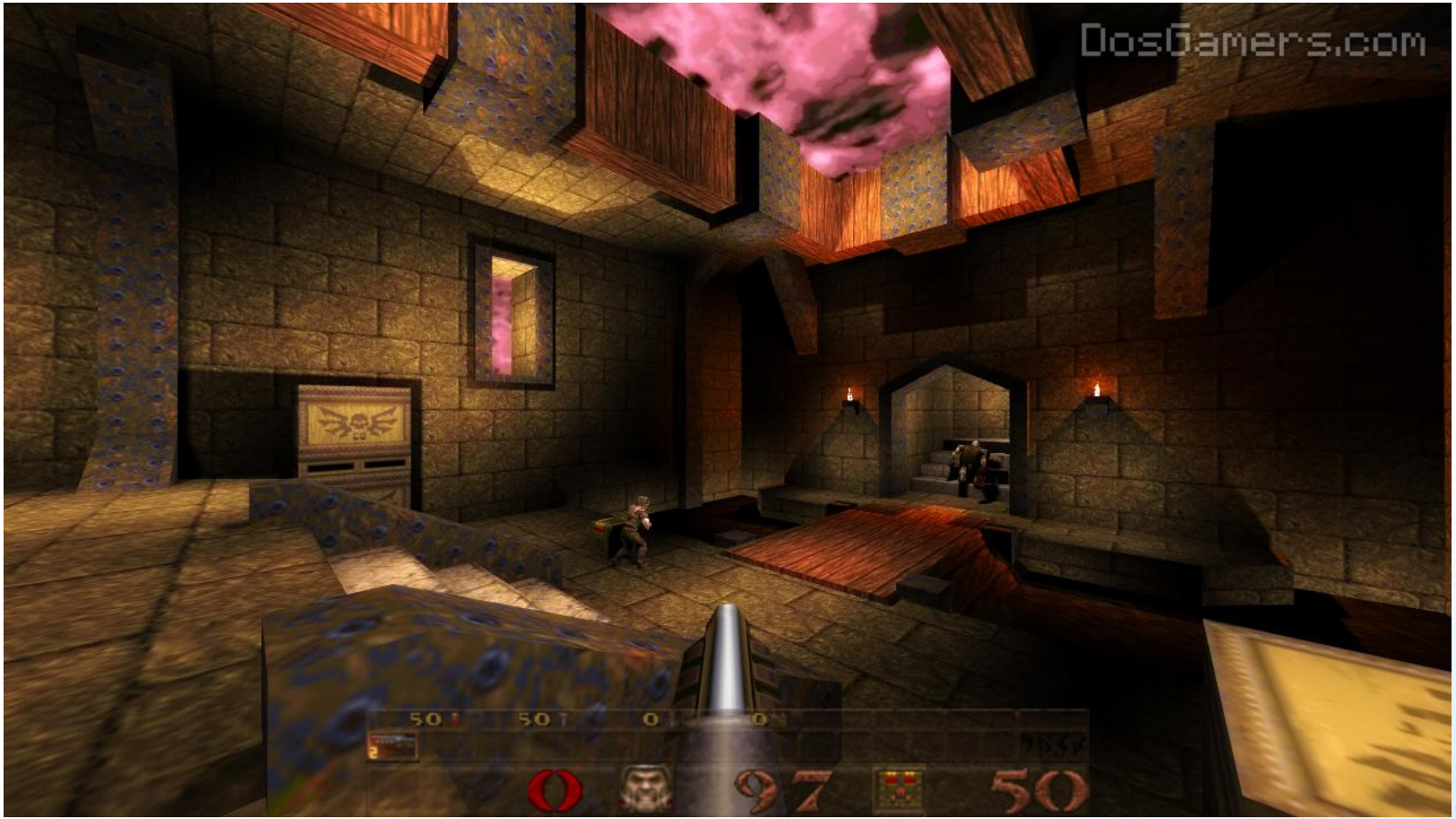
Realtime Rendering

- Realtime Rendering aim is to produce the best possible "visual approximation" of the reality given the processing power of the target hardware.
- Target for eye-perception of smoothness is considered > 24 fps. Today modern videogames have a fruition fps range of [30, 60] fps.
- That means that each image shall be generated in [16.667, 33.334] ms. Without accounting for physics … and Audio … and AI …
- 1996: 3dfx Vodoo 1 is the first non-professional GPU for consumer market.
- Real technology booster for CPU and GPU hardware.
- Commonly used in Videogames, Augmented and Virtual Reality, rendering in desktop and mobile applications (more than you imagine).



Battlezone (1980)

Alone in the Dark (1992)



Quake (1997)

Doom 3 (2004)

Crysis 2 (2011)

Deus Ex: Mankind Divided (2016)

How To Do It?

3D Graphics is like a **Stop Motion** Movie

- Someone creates the **Models (**characters and objects) and put them on a shelf.

- The director, for each scene:
  - Set up the scene background and lights.
  - Takes each needed model from the shelf, one by one.
  - Positions each model in a specific location in the set (**World**) rotating it and posing it in the starting position
  - For each shot, the pose of the models is slightly changed to give the impression of movement according to the Model skeleton and the surrounding objects.
  - Each shot is captured by the camera that "sees" only a specific area of the World from a certain angle (**View**).
  - The captured image is projected on the director screen (**Projection**).
  -
- The models are put back on the shelf and another scene is prepared.

Shaun The Sheep

How To Do It?

Realtime Rendering on PC is possible today through the following abstractions

How To Do It?

Realtime Rendering on PC is possible today through the following abstractions

- **Very High Level**

How To Do It?

Realtime Rendering on PC is possible today through the following abstractions

- **Very High Level**

- **High Level**

How To Do It?

Realtime Rendering on PC is possible today through the following abstractions

- **Very High Level**

- **High Level**

- **Medium Level**

How To Do It?

High level abstraction libraries allows to interact with the GPU using rendering instructions that are able to hide the complexity of memory allocation, synchronization, bus usage ...
Focus is on creating "draw" states for different entities in the world.

How To Do It?

High level abstraction libraries allows to interact with the GPU using rendering instructions that are able to hide the complexity of memory allocation, synchronization, bus usage …

Focus is on creating "draw" states for different entities in the world.

- Library Specification
- Only Computer Graphic
- Portable (Win, Linux, Mobile, …)
- Right Hand Convention
- Column-Major Convention
- Speed and Quality depends on implementation

How To Do It?

High level abstraction libraries allows to interact with the GPU using rendering instructions that are able to hide the complexity of memory allocation, synchronization, bus usage …
Focus is on creating "draw" states for different entities in the world.

- Library Specification
- Only Computer Graphic
- Portable (Win, Linux, Mobile, …)
- Right Hand Convention
- Column-Major Convention
- Speed and Quality depends on implementation

- Library Proprietary Implementation
- Computer Graphic, Audio, Input …
- Only Windows
- Left Hand Convention
- Row-Major Convention
- Speed and Quality is well-known

How To Do It?

High level abstraction libraries allows to interact with the GPU using rendering instructions that are able to hide the complexity of memory allocation, synchronization, bus usage …
Focus is on creating "draw" states for different entities in the world.

⇐ Our Choice

- Library Specification
- Only Computer Graphic
- Portable (Win, Linux, Mobile, …)
- Right Hand Convention
- Column-Major Convention
- Speed and Quality depends on implementation

- Library Proprietary Implementation
- Computer Graphic, Audio, Input …
- Only Windows
- Left Hand Convention
- Row-Major Convention
- Speed and Quality is well-known

Geometric Tools- 3D Vectors

Vectors are one of the most important concepts in 3D graphics They are used to express:

- 3D Locations
- Translation
- Distances
- Rays
- Coordinate System
- Direction

Dot Product

A * B = ||A|| * ||B|| * cos(AngleAB)

Memberwise multiplication, yelds a scalar value.

If unit vectors, the cosine is used to check if the vectors are parallel or orthogonal for example

Cross Product

A x B = (AyBz – AzBy)
(AzBx – AxBz)
(AxBy – AyBx)

Memberwise multiplication, yelds a vector orthogonal to the source vectors.

Geometric Tools- 3D Matrix

3D Matrices are used to express RotoScaling Translations. They are commonly used in Homogeneous format because translation requires an additional column. This reflects on vectors too (from [x,y,z] to [x,y,z,w]).

Scale

$$\begin{pmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Combined through *multiplication* in SRT form.
Order of operations is really important!.

Translation

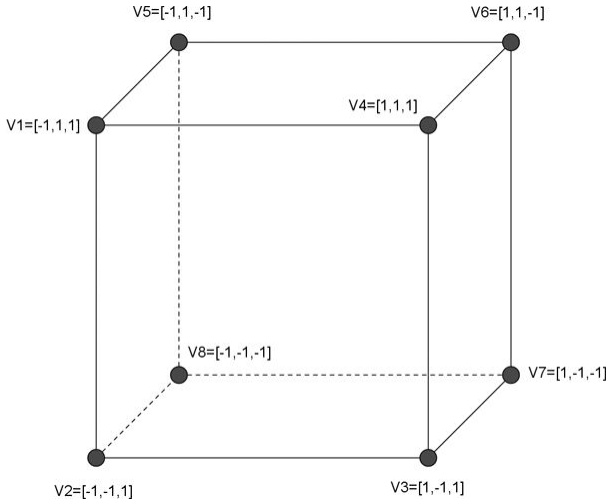
$$\begin{pmatrix} 0 & 0 & 0 & Tx \\ 0 & 0 & 0 & Ty \\ 0 & 0 & 0 & Tz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotations

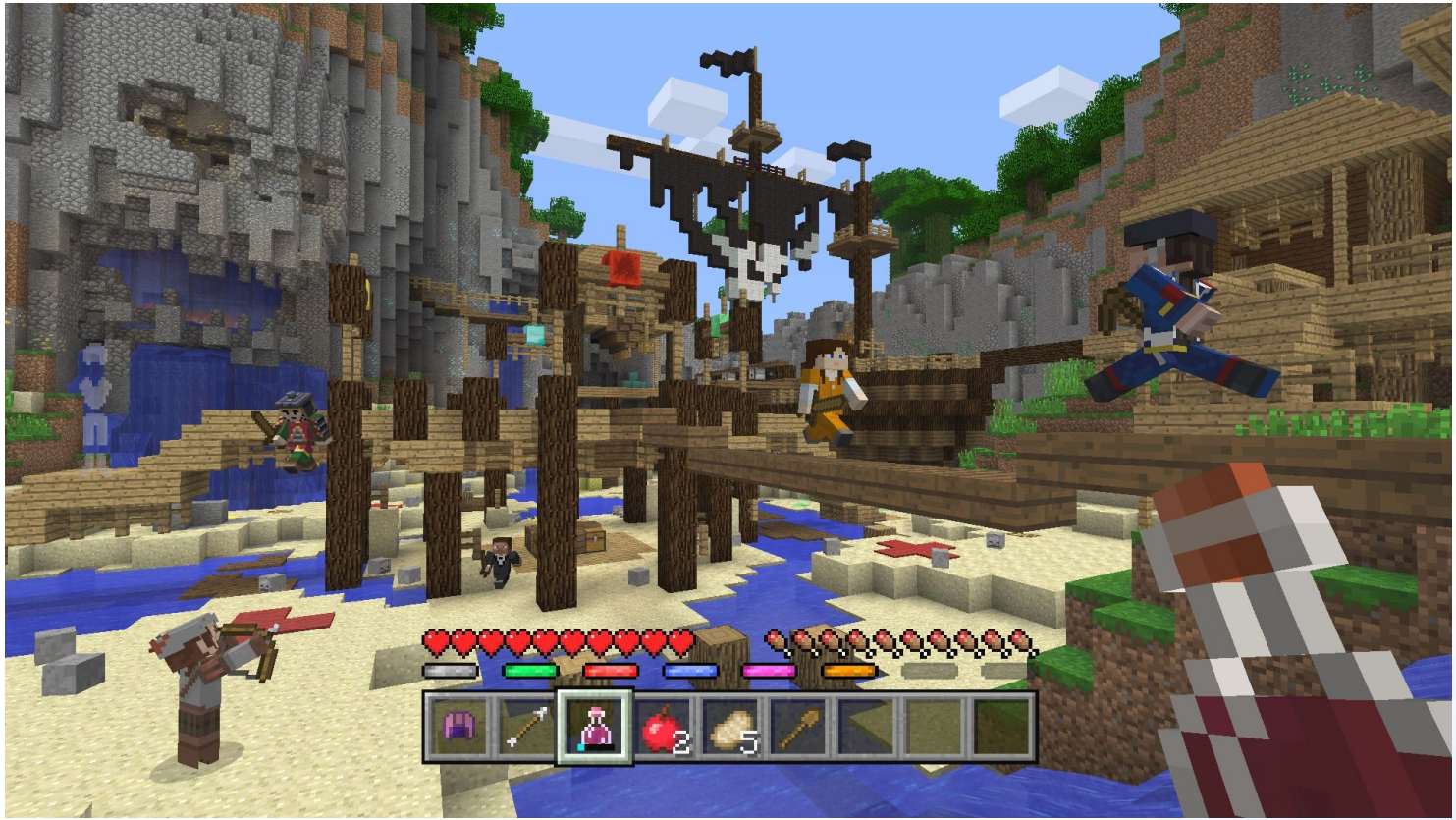
$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Our Beautiful Model

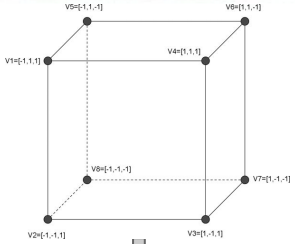
This is our cube companion, the basic 3D model used by everyone when learning 3D Graphics.



"But this model is boring and useless" ....



MineCraft – Bought by Microsoft for 2.5 Billion $



Vertex Data

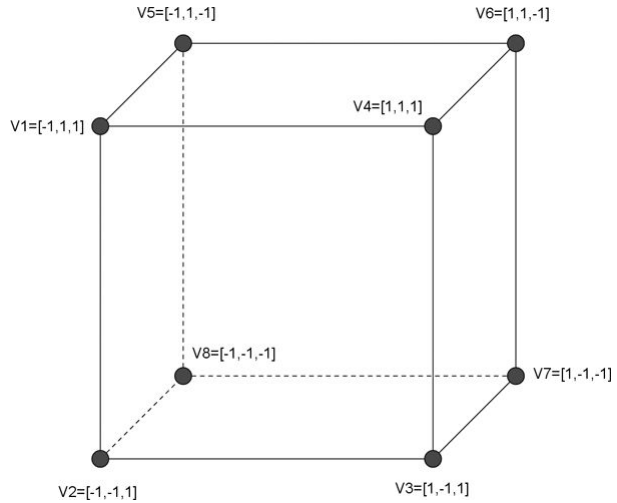
The Graphic Pipeline

Data Preparation

Vertex data is extracted from the model and is put into Vertex Buffers sent to the GPU

Data Preparation – Flat Colored Cube

Let's define it for OpenGL consumption. This vertex data is commonly loaded from a file exported from an authoring tool like Blender, Maya, etc …

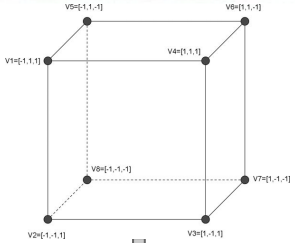


Vertex Data
Position and Color
Counter Clockwise

```
std::vector<GLfloat> cube_vertex =
{
  -1.0f, -1.0f, -1.0f,      1.0f, 0.0f, 0.0f,
   1.0f, -1.0f, -1.0f,      1.0f, 0.0f, 0.0f,
   1.0f,  1.0f, -1.0f,      1.0f, 0.0f, 0.0f,
   1.0f,  1.0f, -1.0f,      1.0f, 0.0f, 0.0f,
  -1.0f,  1.0f, -1.0f,      1.0f, 0.0f, 0.0f,
  -1.0f, -1.0f, -1.0f,      1.0f, 0.0f, 0.0f,

  -1.0f, -1.0f,  1.0f,      0.0f, 1.0f, 0.0f,
   1.0f, -1.0f,  1.0f,      0.0f, 1.0f, 0.0f,
   1.0f,  1.0f,  1.0f,      0.0f, 1.0f, 0.0f,
   1.0f,  1.0f,  1.0f,      0.0f, 1.0f, 0.0f,
  -1.0f,  1.0f,  1.0f,      0.0f, 1.0f, 0.0f,
  -1.0f, -1.0f,  1.0f,      0.0f, 1.0f, 0.0f,

  ...
};
```

Our Beautiful Model – Flat Colored Cube

```
GLuint VBO, VAO;

glGenVertexArrays(1, &VAO);          // Create Buffers on the GPU for the Vertex Array Object
glGenBuffers(1, &VBO);               // and the Vertex Buffer Object

glBindVertexArray(VAO);              // From now on we're operating and configuring the bound VAO (State Machine!)

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(cube_vertex), cube_vertex, GL_STATIC_DRAW);

// Tell Opengl were to find the Position attributes in the configured array
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);

// Tell Opengl were to find the Color attributes in the configured array
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(1);

glBindVertexArray(0);                // The VAO state is closed. The VAO now contain all the previous steps.
```

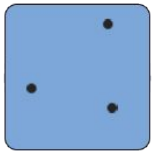


The Graphic Pipeline

Vertex Data

Vertex Shader

Vertices are manipulated in parallel with the Model-View-Projection Transformation to position themselves in the desired "pose"



Vertex Shader – Model Positioning

To position the model in the correct final pose, the **MVP** (Model-View-Projection Matrix) shall be generated in each frame combining Rotations, Scaling and Translations. The final matrix shall then be passed to the vertex shader so it can be applied in parallel to all model vertices.
When in the Clip Space, all vertices still visible (in range [-1.0. 1.0] are converted in Screen Space according to the desired resolution.

Model

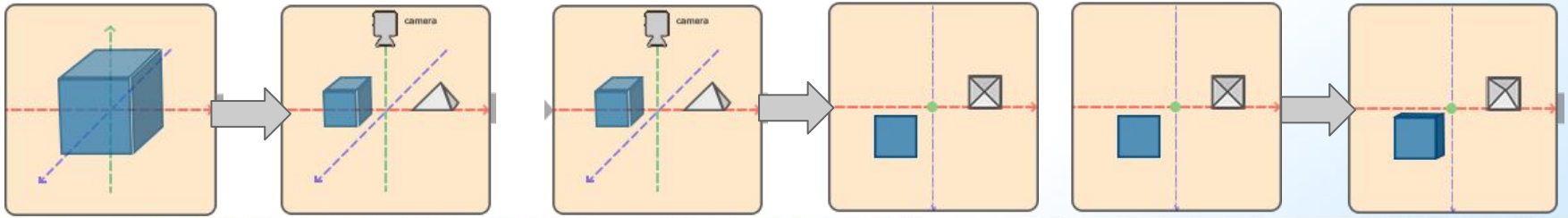
Transforms the model from its original coordinate space to the **World**-relative position.

View

Transforms the model from the **World** to the **Camera**-relative position.

Projection

Transforms the model from the **Camera** to the **Projection** space position (also called Clip Space).



Vertex Shader – What Is It?

- **Shader**: small program written in the shader language compatible with the used GL (**OpenGL has GLSL**)
- Run in parallel on each GPU core for each vertex of the buffer currently bound.
- Has built-in types for vertex manipulation.
- Mandatory output of the Vertex Clip Space position.
- Can take inputs from the ***generic*** code and pass values to the next shaders.

```
#version 330 core
layout (location = 0) in vec3 position;      // Input Vertex Position -> from current state attribute
layout (location = 1) in vec3 in_color;      // Input vertex color -> from current state attribute

uniform mat4 model_mat;                      // Input Model->World Transformation Matrix
uniform mat4 view_mat;                       // Input World->Camera Transformation Matrix
uniform mat4 projection_mat;                 // Input Camera->Clip Transformation Matrix
out vec3 vert_shader_color;                  // Color is passed to the fragment shader as it is

void main()
{
   gl_Position = projection_mat * view_mat * model_mat * vec4(position, 1.0f);      // Mandatory gl_Position
   vert_shader_color = in_color;             // Propagate Vertex Color
}
```

Vertex Shader – Compiling

- All shaders shall be compiled into a **Shader Program**
- When a Shader Program is activated, all draw operations will use its shaders until it deactivated.

```
const GLchar* vertexShaderSource = " ... shader code... ";
const GLchar* fragShaderSource = " ... shader code... ";

// Compile Vertex Shader
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
// Here we'll have the code for checking compilation errors
// Do The Same For Fragment Shader

// Link shaders
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
// Here we'll have the code for checking linking errors

glDeleteShader(vertexShader);              // Clean Up
glDeleteShader(fragmentShader);
```

Vertex Shader – Passing Data

- Data can be passed through Attributes

```
// Tell Opengl were to find the Position attributes in the configured array
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);

// Tell Opengl were to find the Color attributes in the configured array
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(1);
```

```
#version 330 core
layout (location = 0) in vec3 position;      // Input Vertex Position -> from current state attribute
layout (location = 1) in vec3 in_color;      // Input vertex color -> from current state attribute
```

Vertex Shader – Passing Data

- Data can be passed through Uniforms

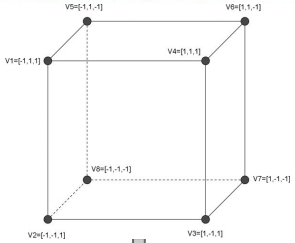

```
// Define the viewport dimensions -> Responsible to Clip Space -> Screen Space conversion
glViewport(0, 0, 800, 600);

// Model -> World: First Scale, then Rotate, finally Translate
glm::mat4 model;
model = glm::scale(model, glm::vec3(0.5f, 0.5f, 0.5f));
model = glm::rotate(model, glm::radians(20.0f), glm::vec3(1.0f, 1.0f, 1.0f));
model = glm::translate(model, glm::vec3(1.0f, 1.0f, 0.0f));

// World -> View: Camera is at (0.0, 0.0, 3.0)
glm::mat4 view;
view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));

// View -> Clip: FoV 45 degree, Frustrum 800x600 Perspective with Near/Far Planes at 0.1/100.0 Z
glm::mat4 projection = glm::perspective(45.0f, (GLfloat)800 / (GLfloat)600, 0.1f, 100.0f);
```

Vertex Shader – Passing Data

- Data can be passed through Uniforms


```
// Retrieve Uniform symbols from Shader Program
GLint modelLoc = glGetUniformLocation(shaderProgram, "model_mat")
GLint viewLoc = glGetUniformLocation(shaderProgram, "view_mat");
GLint projLoc = glGetUniformLocation(shaderProgram "projection_mat");

// Bind the matrices to the shader symbols
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));
```

```
uniform mat4 model_mat;          // Input Model->World Transformation Matrix
uniform mat4 view_mat;                  // Input World->Camera Transformation Matrix
uniform mat4 projection_mat;     // Input Camera->Clip Transformation Matrix
```

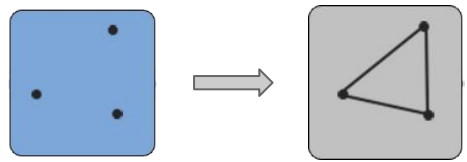


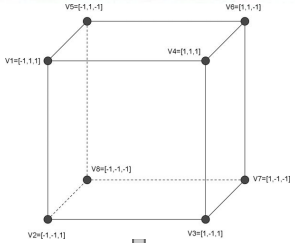
The Graphic Pipeline

Vertex Data

Shape Composition

The vertices are linked with lines to compose shapes



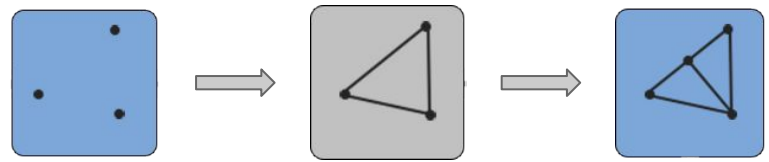


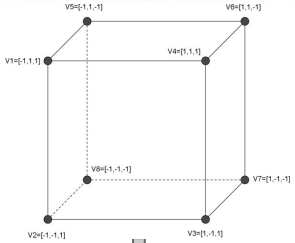
The Graphic Pipeline

Vertex Data

Geometry Shader (Optional)

The shape can be subdivided into subshapes (useful for effects like tessellation)



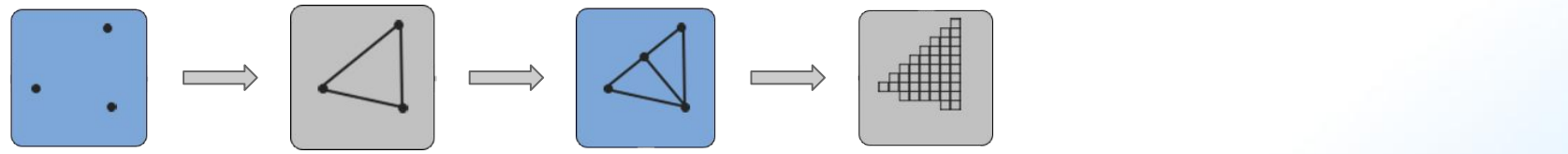


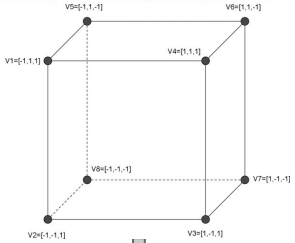
The Graphic Pipeline

Vertex Data

Rasterization

The "smooth" shape is converted into its pixel-equivalent representation. Each drawable unit is called **fragment**.



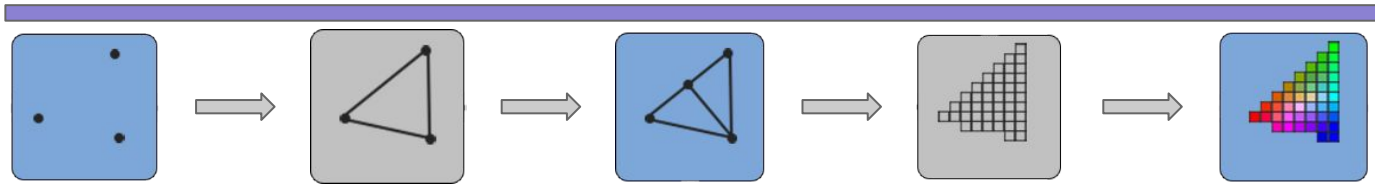


The Graphic Pipeline

Vertex Data

Fragment Shader

Fragments are manipulated in parallel to determine their final color. This is the stage where most visual effects are applied (lighting, shadows, ...)

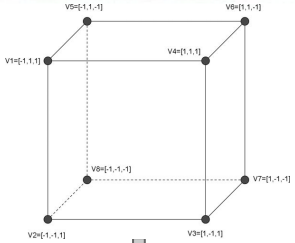


Fragment Shader

- Our fragment shader will output the color passed by the Vertex Shader

```
#version 330 core
…
out vec3 vert_shader_color;                    // Color is passed to the fragment shader as it is

void main()
{
   gl_Position = projection_mat * view_mat * model_mat * vec4(position, 1.0f);      // Mandatory gl_Position
   vert_shader_color = in_color;               // Propagate Vertex Color
}
```

```
#version 330 core

in vec3 vert_shader_color;
out vec4 color;

void main()
{
   color = vec4(vert_shader_color, 1.0f);    // Alpha channel as 1.0
}
```

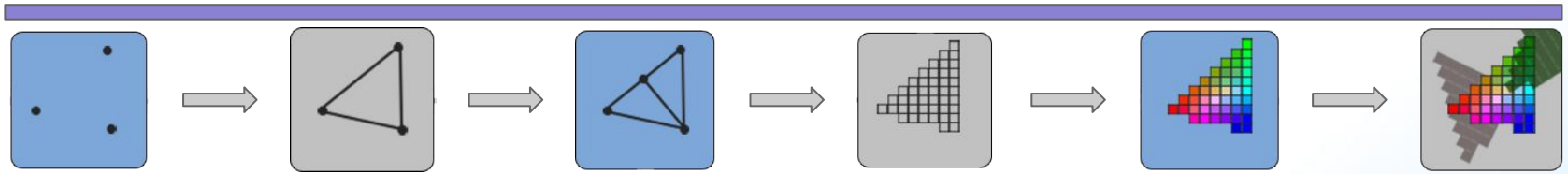


The Graphic Pipeline

Vertex Data

Test and Blending

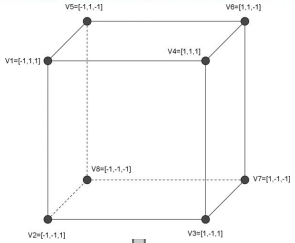
Discards all pixels that are not visible due to occlusion. Modify the final pixel color according to alpha values.



Z-Buffer and Depth Testing

- The Z-Buffer is an OpenGL buffer large as the viewport where the Z values of the nearest fragments are kept.
- If a new fragment has a Z nearer to the currently stored one for the specific (x,y) position, the fragment is drawn in the framebuffer and the Z value is updated, otherwise the fragment is discarded.

```
// Initialization Code
…
glEnable(GL_DEPTH_TEST);
…

// In Drawing Code add the Depth Buffer reset
…
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
…
```

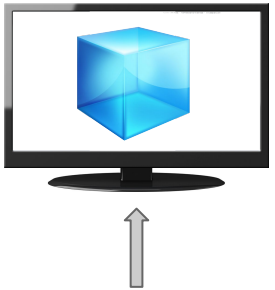


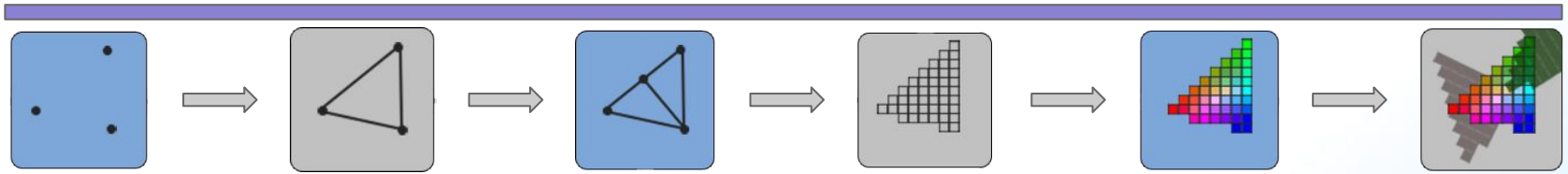
The Graphic Pipeline

Vertex Data

Visualization

Composed image in the framebuffer is ready to be displayed.



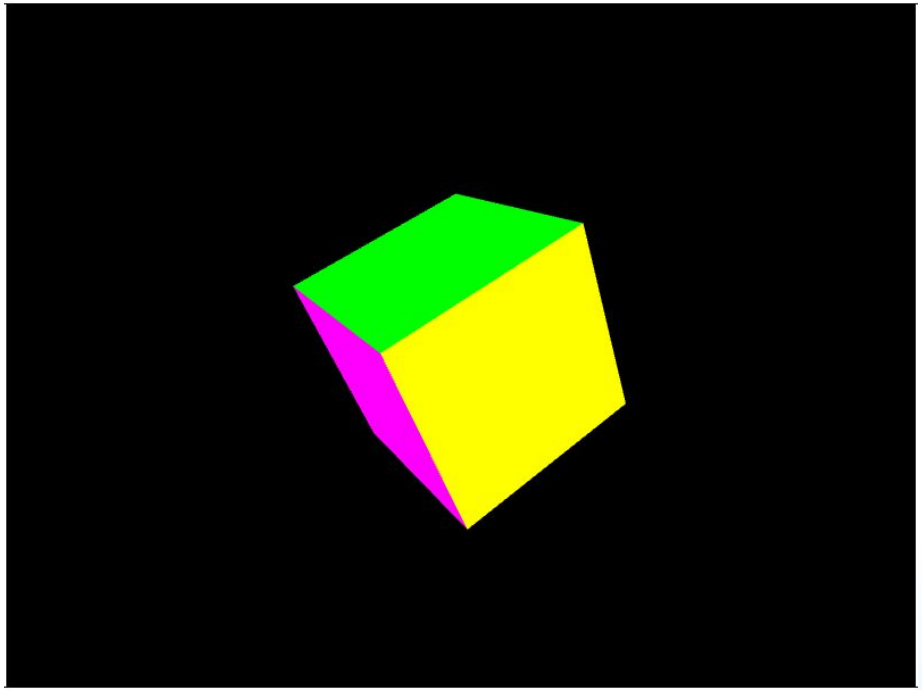


Framebuffer Switch

- All fragments for the current frame are put in the currently active FrameBuffer.
- In **Double Buffering** while one buffer is being drawn on, the other one is used by the GPU to put pixels on screen. Then buffers are switched and a new frame is generated.
- The Framebuffer Switch command depends of the library/code used to manage the window.

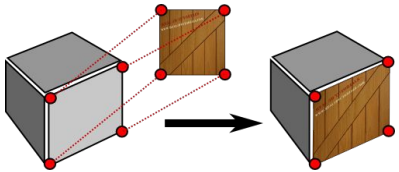
```
// Basic Frame Drawing Loop
while(loop)
{
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUseProgram(shaderProgram);
    glBindVertexArray(VAO);

    glDrawArrays(GL_TRIANGLES, 0, 36);    // Draws triangles based on the first 36 vertex from the currently bound VAO program
                                          // and using the active shader
    glBindVertexArray(0);

   // Swap the screen buffers
   SDL_GL_SwapWindow(window);   // Dependent of the Window Manager used (SDL, GLFW, etc....)
}
```

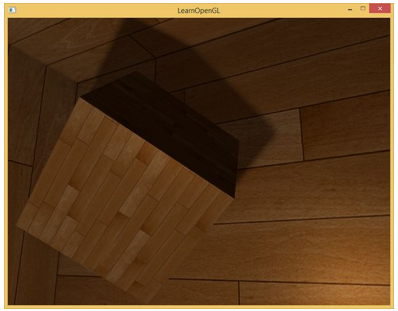
Finally the result of hard work!

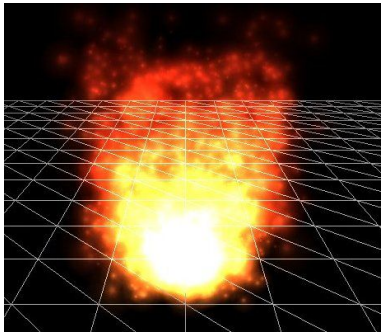


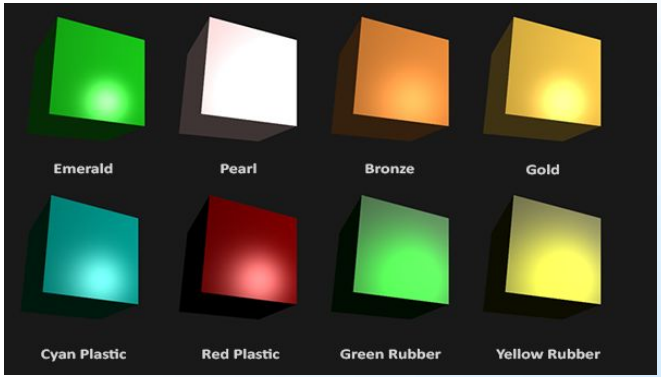
Where to go from here?

- Just scratched the surface of a wide wide wide world
  - Texture Mapping
  - Lighting
  - Shadows
  - Skeletal Animation
  - Particle Effects
  - Terrain Rendering, height maps
  - And much much more



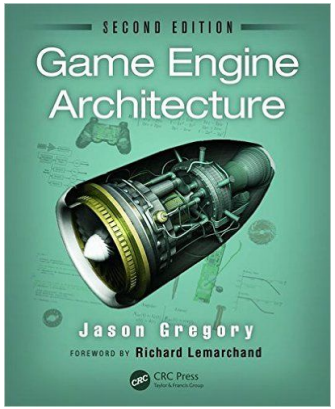


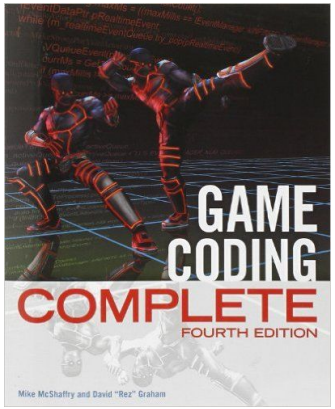




Resources

- **Internet has all you need**: CG is one of the most requested topic, lots of tutorial
- www.learnopengl.com
- **Some Good Books available**





THANK YOU !!!!