

Peter Mileff PhD

# Programming of Graphics

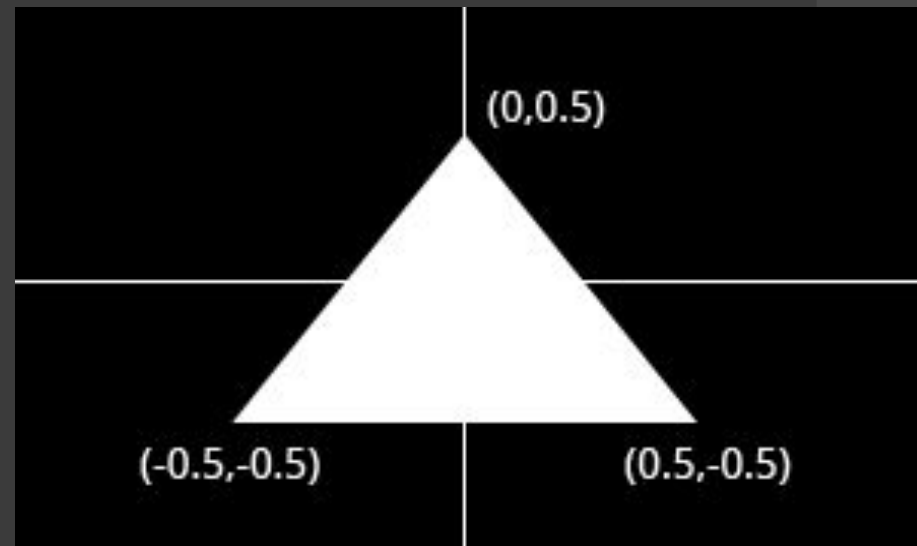
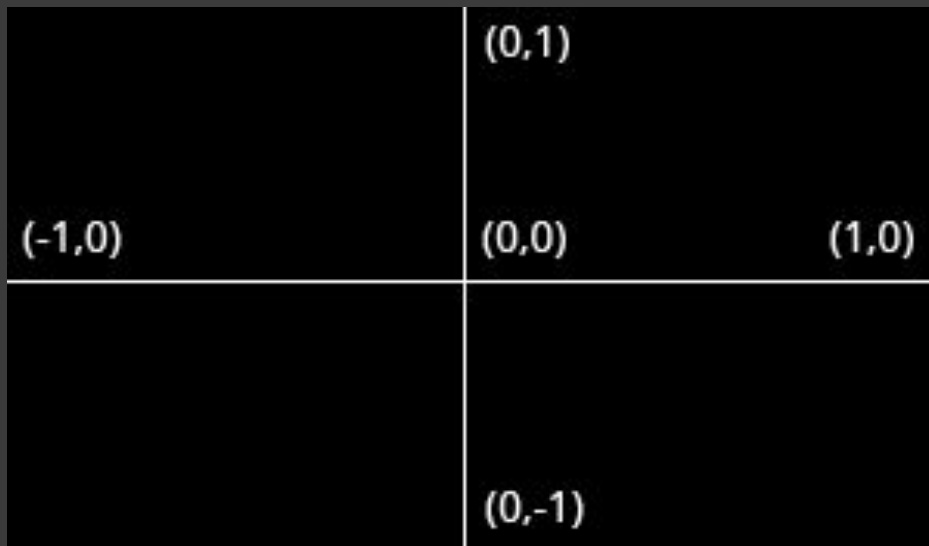
**Introduction to 2D graphics**

University of Miskolc  
Department of Information Technology

# The coordinate system...

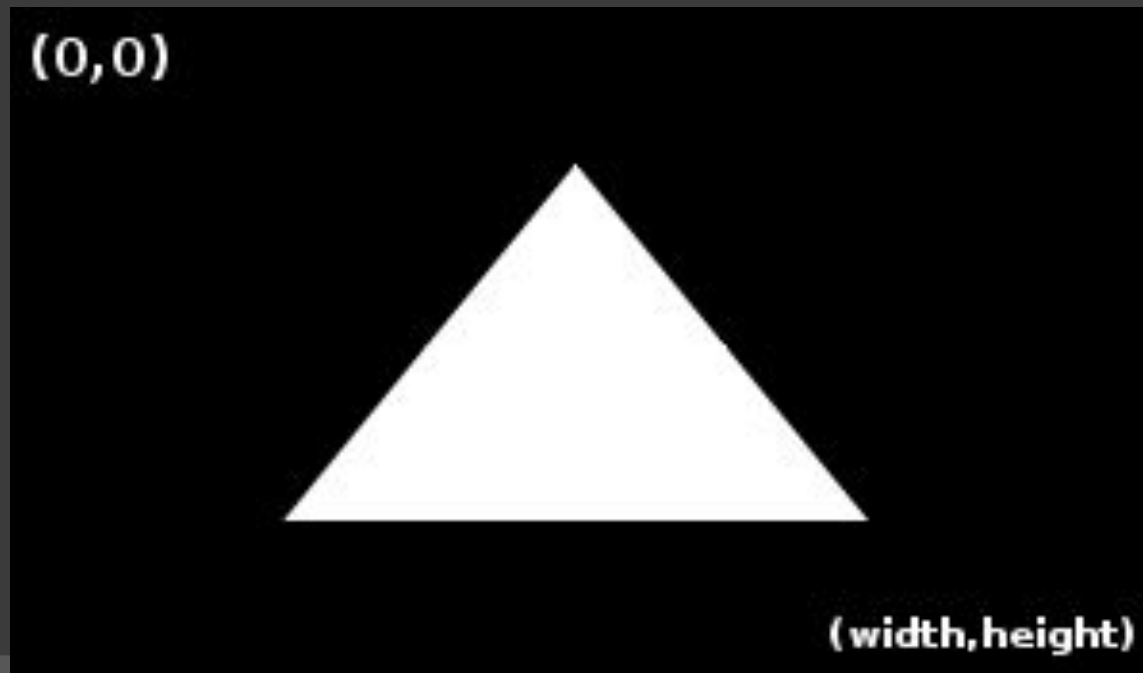
# The OpenGL coordinate system in 3D

- When vertices have been processed by the pipeline, their coordinates will have been transformed into device coordinates.
- Device X and Y coordinates are mapped to the screen **between -1 and 1**
- Any coordinates that fall outside this range will be discarded/clipped and won't be visible on your screen

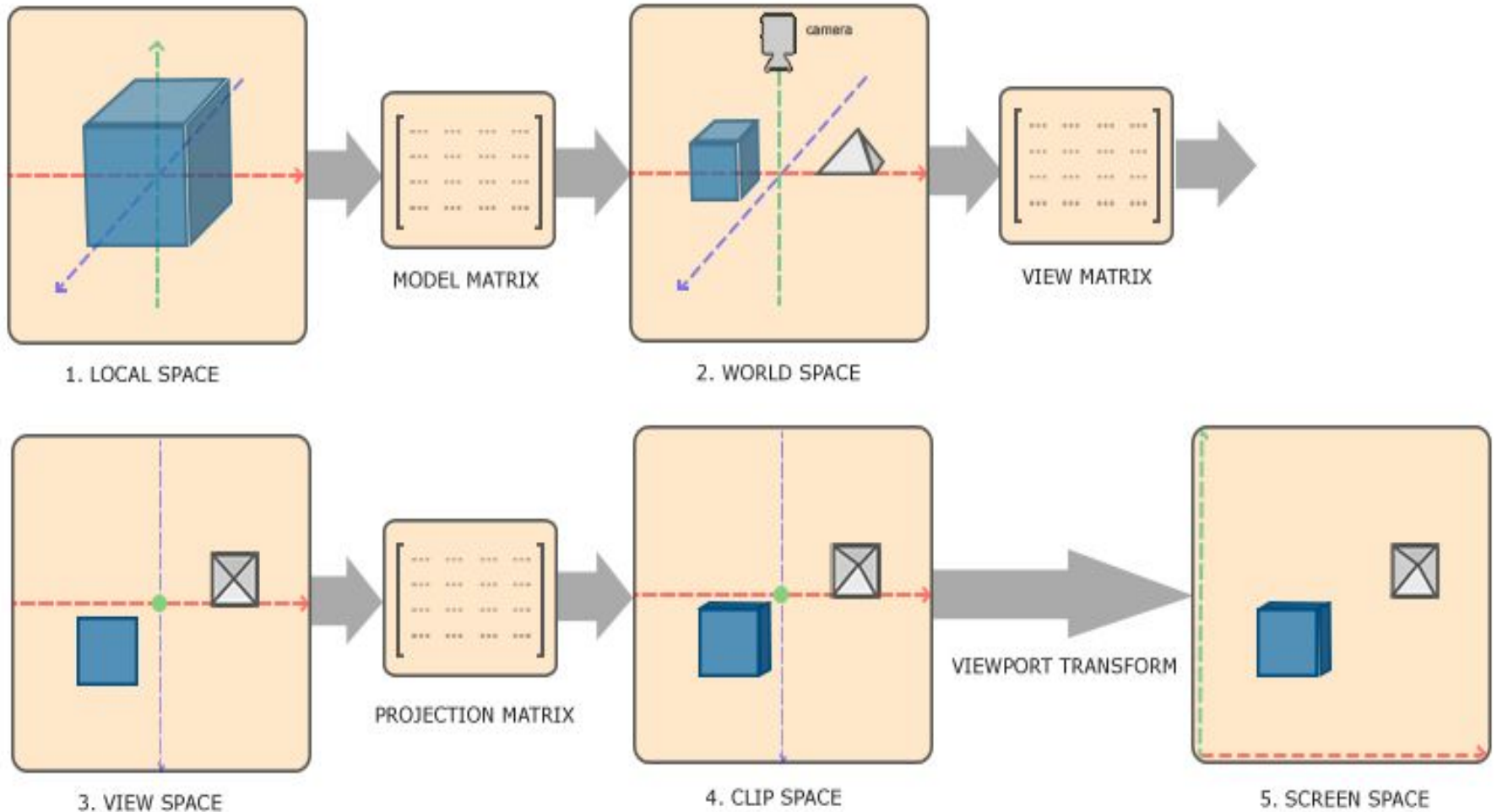


# The OpenGL coordinate system in 2D

- In 2D, the coordinate system is usually “easier” to understand
- Mostly used mapping:
  - Screen top-left (or top-bottom) is the origo
  - The other corner of the screen matches the window dimensions



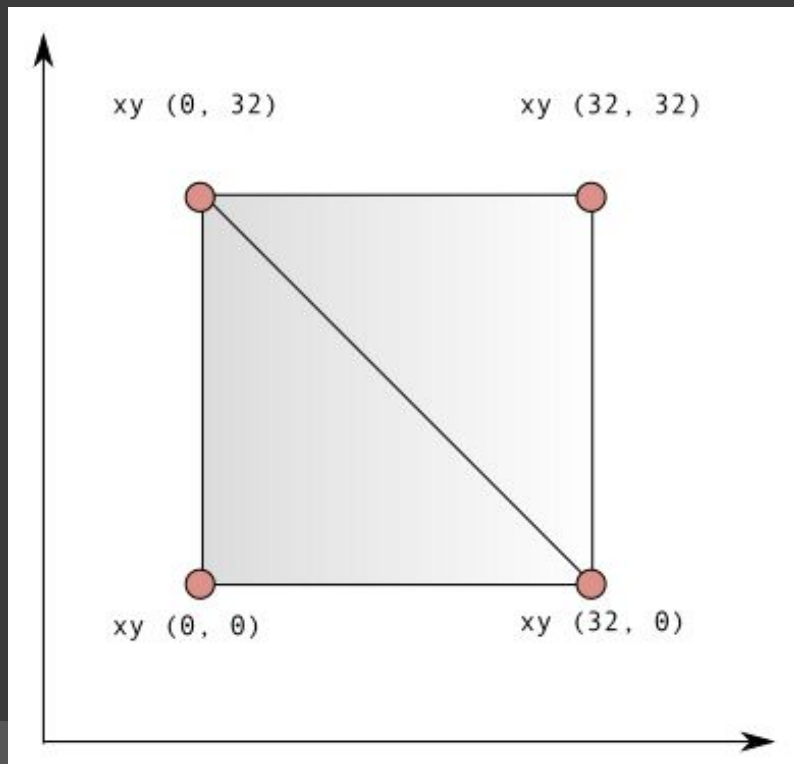
# The global picture



**Drawing a quad to the  
screen in 2D...**

# Drawing a quad

- To start drawing something we have to first give OpenGL some input vertex data
  - Data may come from files
  - Or data can be constructed by the program



Vertices = {

0.0, 0.0, 0.0,  
0.0, 32, 0.0,  
32, 0.0, 0.0,

32, 0.0, 0.0,  
0.0, 32, 0.0,  
32, 32, 0.0,

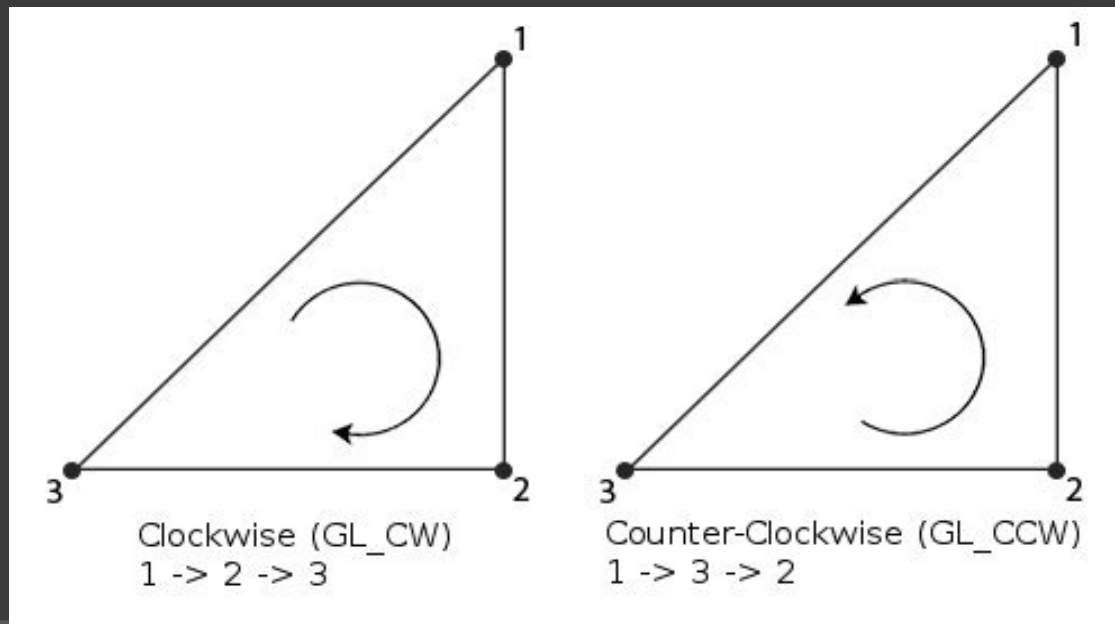
}

**Triangle 1**

**Triangle 2**

# The OpenGL coordinate system in 3D

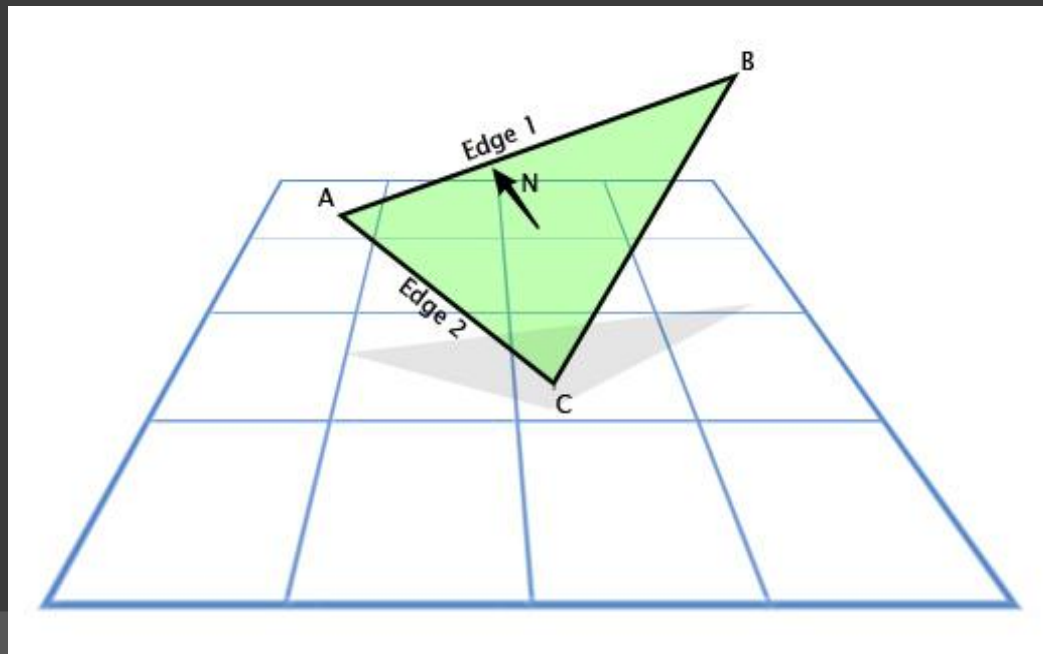
- **The order of vertices is very important!**
- This order can be:
  - Clockwise
  - Counter-clockwise





# The OpenGL coordinate system in 3D

- Winding order is used to
  - calculate the triangle normal vector,
  - decide that the triangle is being seen from the "front" or the "back" side
    - Primary usage is to culling out the back faced geometry



# Prepare the data

- Prepare the OpenGL to draw:
  - The vertex data should transfer into the GPU memory
  - A memory buffer should be created on the GPU
- Modern OpenGL solutions are:
  - Vertex Buffer Object (VBO)
  - Vertex Array Object (VAO)
- The advantage of using those buffer objects:
  - we can send large data all at once to the graphics card without having to send data a vertex a time
  - Once the data is in the graphics card's memory the vertex shader has access to the vertices
    - making it extremely fast

# Prepare the data

- Preparing the data has the following steps:

- 1. Create a buffer: this buffer has a unique

```
int vboID = glGenBuffers();
```

- 2. Bind the buffer: after we can use it

```
glBindBuffer(GL_ARRAY_BUFFER, vboID);
```

OpenGL has many types of buffer objects. The buffer type of a vertex buffer object is `GL_ARRAY_BUFFER`

# Prepare the data

- 3. Transfer data to th GPU memory

In java:

```
glBufferData(GL_ARRAY_BUFFER, verticesBuffer,  
            GL_STATIC_DRAW);
```

In C/C++:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(verticesBuffer),  
            verticesBuffer, GL_STATIC_DRAW);
```

GL\_STATIC\_DRAW: the data will most likely not change at all or very rarely (e.g. the world)

GL\_DYNAMIC\_DRAW: The vertex data will be created once, changed from time to time, but drawn many times more than that

GL\_STREAM\_DRAW: the data will change every time it is drawn

This usage value will determine in what kind of memory the data is stored on your graphics card for the highest efficiency


# Shaders

- Because of the modern pipeline, we should write shaders:
  - Vertex and fragment shaders

Position vertex  
attribute

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```



# Shaders

- The fragment shader:

The color of the pixel

```
#version 330 core  
out vec4 FragColor;
```

```
void main()
```

```
{
```

```
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
```

```
}
```

R

G

B

A



# Compiling the shaders

- In order to use the shaders, a **shader program** should be created:
  - Steps:
    - Compile each shaders
    - Link shaders into a shader program

# Compiling the shaders

- We compile shaders at run-time from its source code
  - The first thing we need to do is create a shader object
  - We store the vertex shader as an (unsigned) int and create the shader with glCreateShader:

```
int vertexShaderID = glCreateShader(GL_VERTEX_SHADER);
```

- We provide the type of shader we want to create as an argument.
  - Since we're creating a vertex shader we pass in `GL_VERTEX_SHADER`



# Compiling the shaders

- Next we attach the shader source code to the shader object and compile the shader:

## In C/C++:

```
glShaderSource(vertexShaderID, 1, &vertexShaderSource, NULL);  
glCompileShader(vertexShaderID);
```

## In LWJGL:

```
glShaderSource(vertexShaderID, vertexShaderSource);  
glCompileShader(vertexShaderID);
```

# Compiling the shaders

- Check shader compilation status:

## In C/C++:

```
int success;
char infoLog[512];
glGetShaderiv(vertexShaderID, GL_COMPILE_STATUS, &success);
if(!success)
{
    glGetShaderInfoLog(vertexShaderID, 512, NULL, infoLog);
    std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" <<
    infoLog << std::endl;
}
```

## In LWJGL:

```
if (glGetShaderi(vertexShaderID, GL_COMPILE_STATUS) == 0) {
    throw new Exception("Error compiling Shader code: " +
        glGetShaderInfoLog(vertexShaderID, 1024));
}
```

# Creating the shader program

- What is it?
  - A **shader program object** is the final linked version of multiple shaders combined
- To use the recently compiled shaders we have to link them to a shader program object
- How to use it?
  - When rendering objects we activate this shader program
    - Everything we draw after it will use this program

# Creating the shader program

- Creating a program is easy:

```
unsigned int shaderProgram;  
shaderProgram = glCreateProgram();
```

- Attach the previously compiled shaders to the program object:

```
glAttachShader(shaderProgram, vertexShader);  
glAttachShader(shaderProgram, fragmentShader);
```

- Link the shaders:

```
glLinkProgram(shaderProgram);
```

# Creating the shader program

- Get the link status:

## In C/C++:

```
glGetProgramiv(shaderProgram, GL_LINK_STATUS, &success);  
  
if(!success) {  
    glGetProgramInfoLog(shaderProgram, 512, NULL, infoLog);  
    ...  
}
```

## In LWJGL:

```
if (glGetProgrami(programId, GL_LINK_STATUS) == 0) {  
    throw new Exception("Error linking Shader code: " + glGetProgramInfoLog(programId,  
    1024));  
}  
  
}
```

**Using the shader...**

# Applying the shader program

- The vertex shader allows us to specify any input we want in the form of vertex attributes
- After shader program and VBO is created:
  - we have to manually specify what part of our input data goes to which vertex attribute in the vertex shader.

In LWJGL:

Attribute 0      3 coordinates  
`glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);`



If you remember our vertex shader:

`layout (location = 0) in vec3 aPos;`

# Applying the shader program

- Using the shader object to draw our quad is very simple:

```
glUseProgram(shaderProgram);
```

```
glBindVertexBuffer(vboID);
```

```
glDrawArrays(GL_TRIANGLES, 0, 6);
```

```
glUseProgram(0);
```

We draw 6  
vertices



Deactivate shader





# Basics of Texture mapping...

# Texture Mapping

- Texture mapping means applying any type of picture on one or more faces (triangles) of a 3D model
  - For example, if we wanted to convert a basic cube into a wooden box, we simply paint wooden panels onto the individual polygons

## Picture = Texture

- A texture can be anything
  - It is often a pattern such as bricks, foliage, barren land, etc
  - Texture adds realism to the scene.
  - Today modern computer games use a lot of high quality textures
    - Needs a lot of GPU memory!

# Uncharted 4 Game



# Rage Game Megatexture technology



**RAGE**

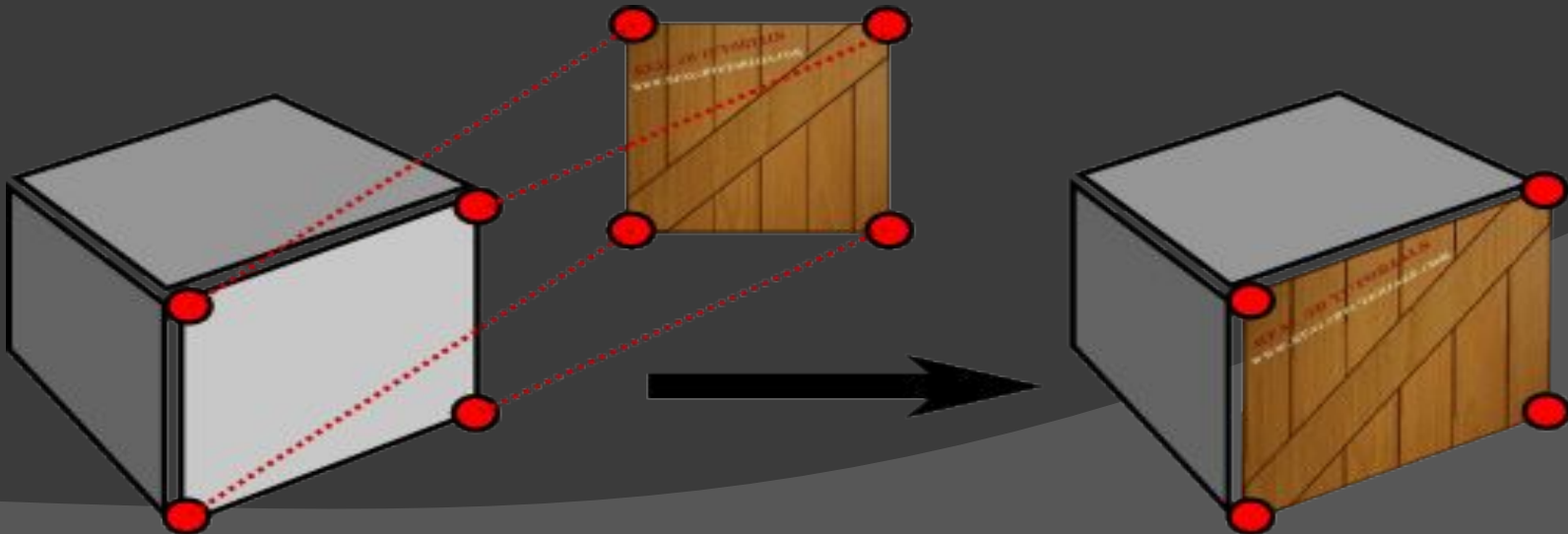
© 2010 id Software LLC. All Rights Reserved.

# Texture Mapping

- OpenGL supports also RGB and RGBA textures
  - But is recommended to create texture images as PNG or TGA files
- PNG / TGA image files have the ability to store alpha values (transparency settings),
  - which is an invaluable feature when creating advanced models and objects.
- ALWAYS ensure that the dimensions of the texture images are in **power of two**
  - e.g. 32x32, 64x64, 128x128, 256x256, 512x512, etc.
  - GPU “likes” these formats only

# How texture mapping works?

- Once we created a texture, the painting of the texture onto polygons is a very straight forward process
  - We simply map the coordinates of the texture to the vertices of the polygon,
  - and OpenGL will automatically map the texture to the polygon



# How texture mapping works?

- A polygon/triangle is usually scaled, rotated and translated
  - During the projection it can land on the screen in numerous ways
  - and look very different depending on its orientation to the camera.
- **What is needed for the GPU?**
  - A texture need to follow the movement of the vertices of the triangle
  - To do this the developer supplies a set of coordinates known as “**texture coordinates**” to each vertex
  - As the GPU rasterizes the triangle it interpolates the texture coordinates across the triangle face

# Texture coordinates

- The question is **how can we specify the texture coordinates of a vertex?**
  - A 2D texture has a width and height that can be any number
  - An universal method needed, which is independent of the texture width and height parameters
    - Because a texture of a model can be changed during the visualisation
- Solution
  - texture coordinates are specified in “**texture space**” which is simply the normalized range  $[0,1]$ .
  - This means that the texture coordinate is usually a fraction
  - By multiplying that fraction with the corresponding width/height of a texture we get the coordinate of the texel in the texture.



# Texture coordinates

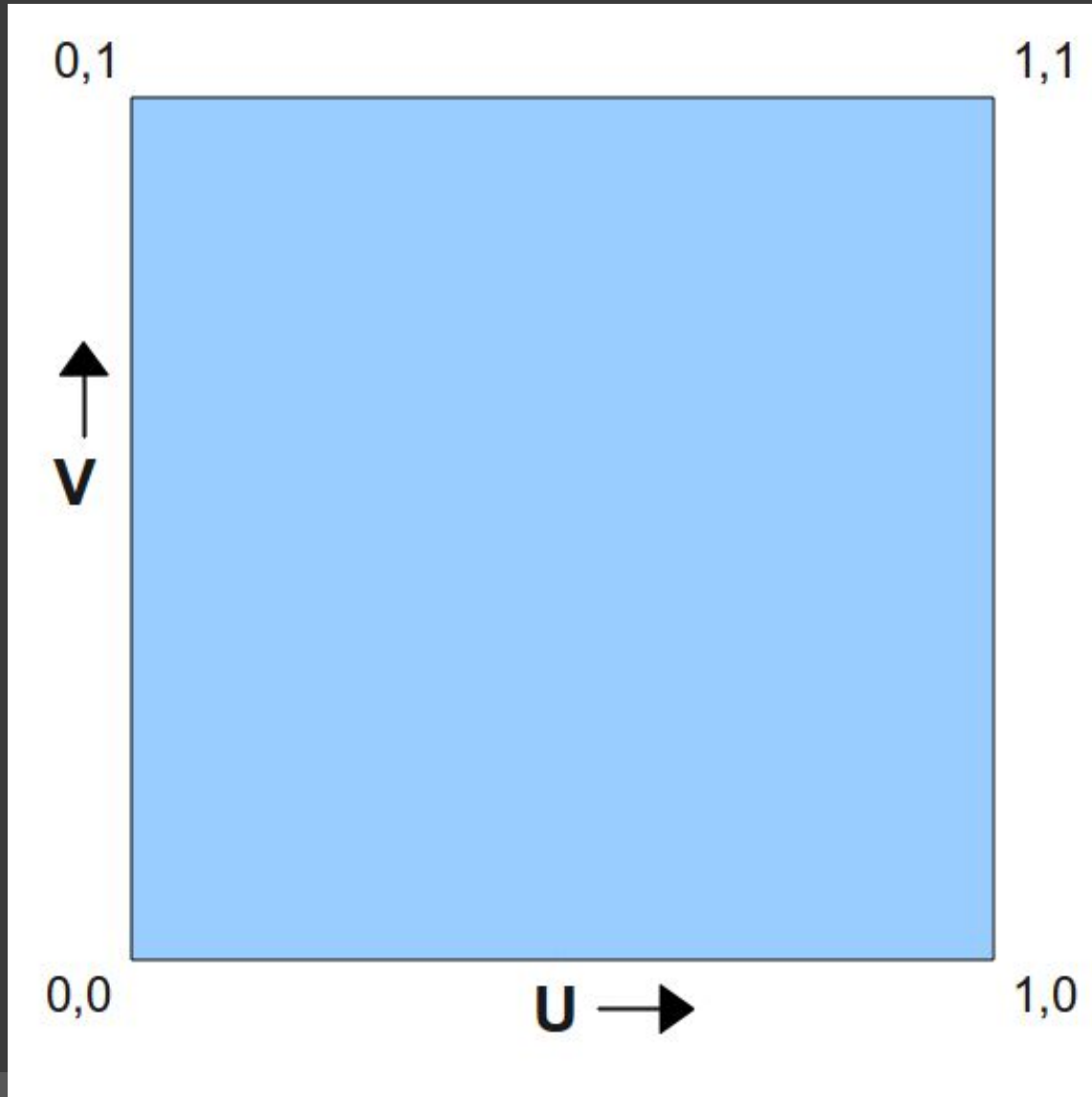
- Example:

- The texture coordinate is [0.5,0.1]
- It has a width of 320 and a height of 200
- The texel location will be (160,20):

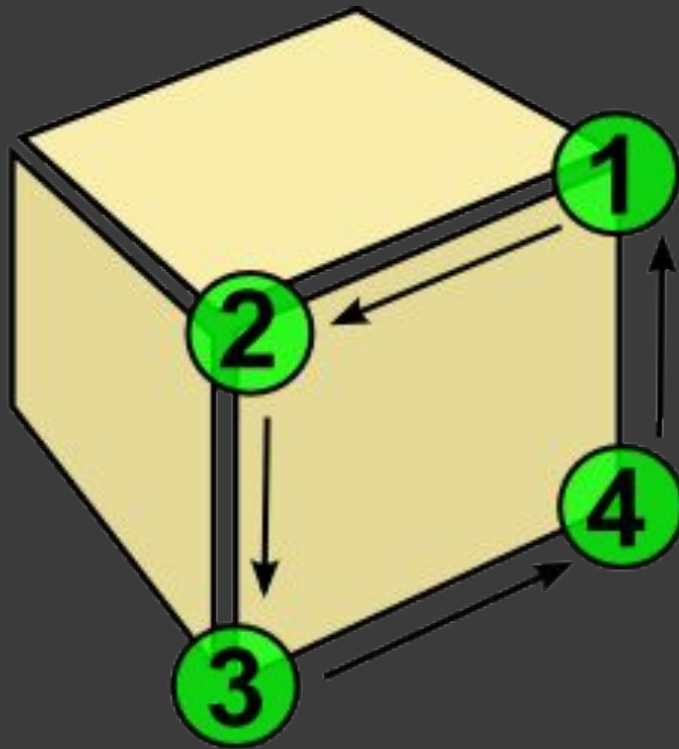
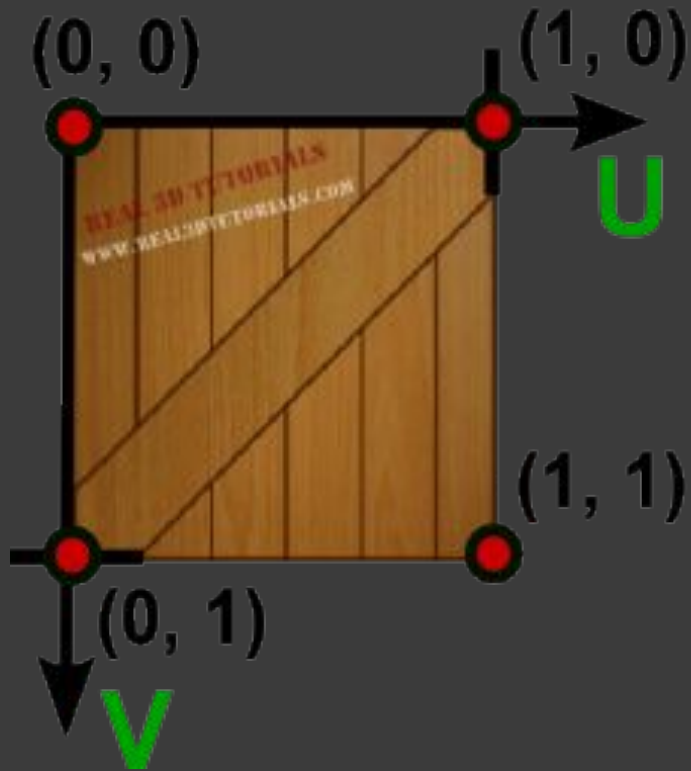
$$0.5 * 320 = \mathbf{160} \text{ and } 0.1 * 200 = \mathbf{20}$$

- The usual convention is to use **U** and **V** as the axis of the texture space
  - U corresponds to X in the 2D cartesian coordinate system and V corresponds to Y
  - OpenGL treats the values of the UV axes as going from left to right on the U axis and down to up on the V axis

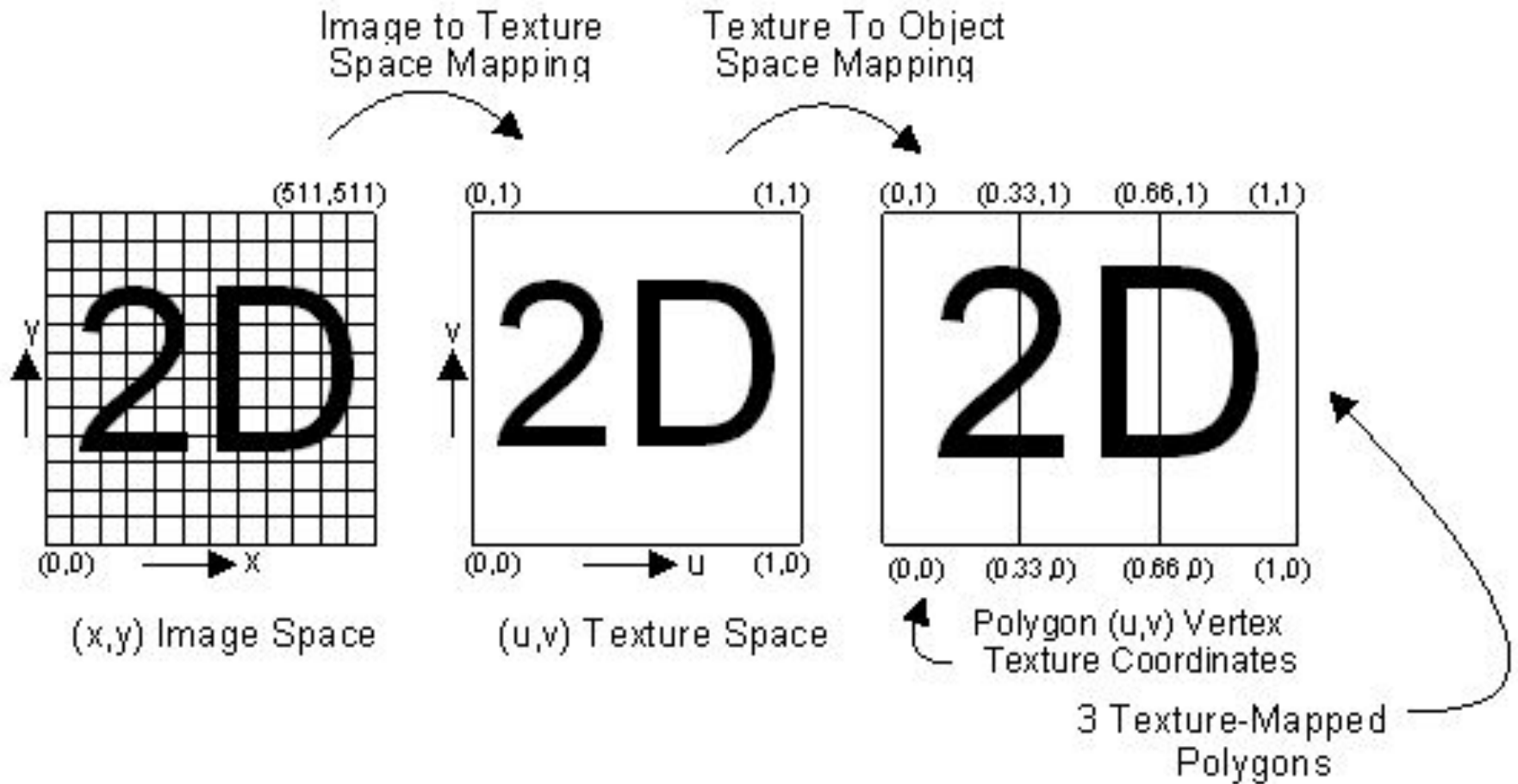
# Texture coordinates



# Texture coordinates



# Texture coordinates



# Texture coordinates

- The texture can be of different types (1D, 2D, etc) with different dimensions
  - the underlying data type can have multiple formats (RGB, RGBA, etc).
- OpenGL provides a way to specify the starting point of the source data in memory
  - and all the above attributes and load the data into the GPU.
- There are also multiple parameters that you can control
  - Such as the filter type
- A texture object is associated with a handle
  - After creating the handle and loading the texture we can simply switch textures on the fly
    - by binding different handles into the OpenGL state.

# Texturing in OpenGL

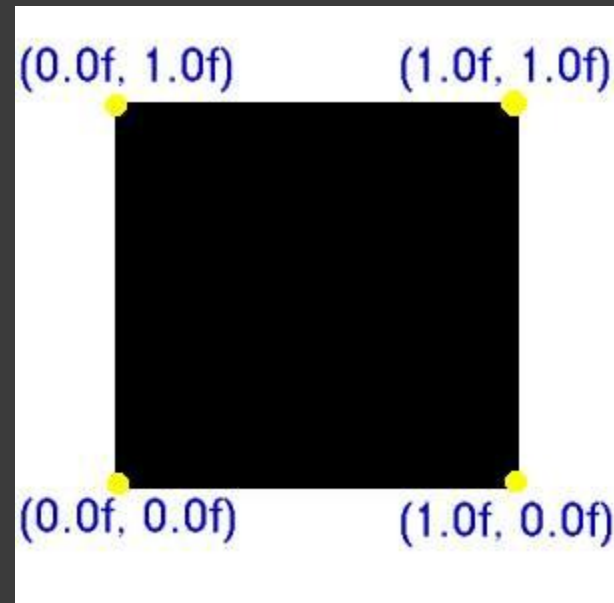
- Define texture coordinate  
`glTexCoord2f(u, v)`
- Enable / disable texturing  
`glEnable(GL_TEXTURE_2D);`  
`glDisable(GL_TEXTURE_2D);`
- Using a texture  
`glBindTexture (GL_TEXTURE_2D, texture_number);`
- Specifying a texture (load into the GPU memory):

```
void glTexImage2D(  
    GLenum target, - e.g: GL_TEXTURE_2D  
    GLint level, - Specifies the level-of-detail number  
    GLint internalFormat, - Specifies the number of color components  
    GLsizei width, - width  
    GLsizei height, - height  
    GLint border, - This value must be 0  
    GLenum format, - the format of the pixel data. E.g.: GL_RGBA  
    GLenum type, - the data type of the pixel data. E.g: GL_UNSIGNED_BYTE  
    const GLvoid * data); - Specifies a pointer to the image data in memory.
```

# Texture coordinates

- An example of texturing a quad (texture is already loaded)

```
glBindTexture (GL_TEXTURE_2D, 13);  
glBegin (GL_QUADS);  
glTexCoord2f (0.0, 0.0);  
glVertex3f (0.0, 0.0, 0.0);  
glTexCoord2f (1.0, 0.0);  
glVertex3f (10.0, 0.0, 0.0);  
glTexCoord2f (1.0, 1.0);  
glVertex3f (10.0, 10.0, 0.0);  
glTexCoord2f (0.0, 1.0);  
glVertex3f (0.0, 10.0, 0.0);  
glEnd ();
```



# Texturing in Modern OpenGL

- We need to define the texture coordinates in a VBO

Example (quad):

```
textCoords = {  
    0.0f, 1.0f, 1.0f,  
    1.0f, 1.0f, 0.0f,  
    1.0f, 0.0f, 0.0f,  
    0.0f, 0.0f, 1.0f };
```

Generate VBO for texture coordinates:

```
vbold = glGenBuffers();  
glBindBuffer(GL_ARRAY_BUFFER, vbold);  
glBufferData(GL_ARRAY_BUFFER, textCoords, GL_STATIC_DRAW);  
glVertexAttribPointer(1, 2, GL_FLOAT, false, 0, 0);
```

 UV attributes location is shader



# Texturing in Modern OpenGL

## The vertex shader:

```
layout (location=0) in vec3 position;  
layout (location=1) in vec2 texCoord;
```

```
out vec2 outTexCoord;
```

```
uniform mat4 worldMatrix;  
uniform mat4 projectionMatrix;
```

```
void main()  
{  
    gl_Position = projectionMatrix * worldMatrix * vec4(position, 1.0);  
    outTexCoord = texCoord;  
}
```

# Texturing in Modern OpenGL

## The fragment shader:

```
in vec2 outTexCoord;  
out vec4 fragColor;
```

```
uniform sampler2D texture_sampler;
```

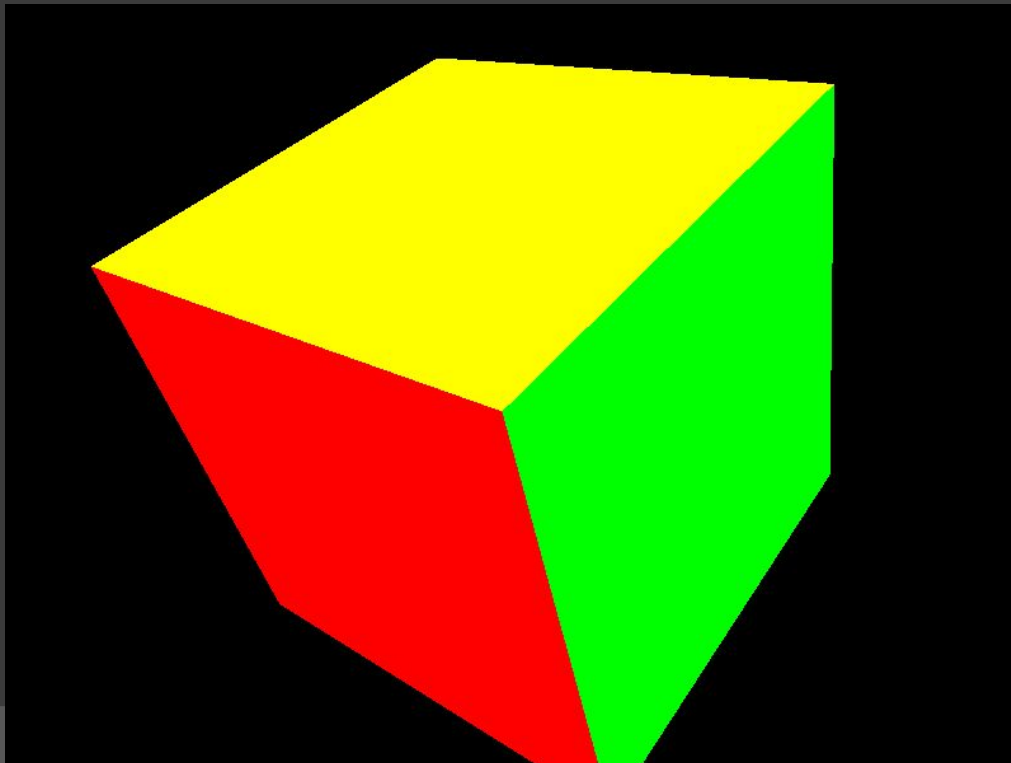
```
void main()  
{  
    fragColor = texture(texture_sampler, outTexCoord);  
}
```

The final color of the pixel

# Buffers in the OpenGL...

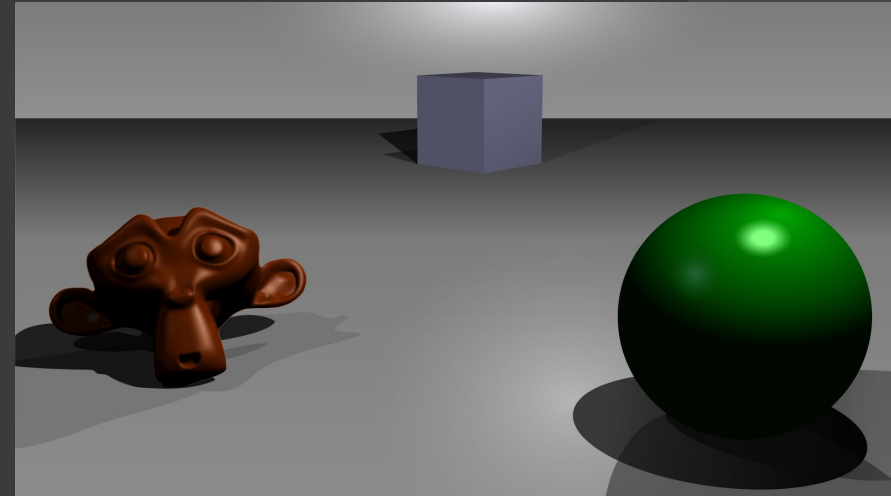
# OpenGL Buffers

- An OpenGL system can manipulate the following Buffers:
- Color buffer: this is the buffer to which we usually draw. Contains the colors in RGBA or indexed mode.

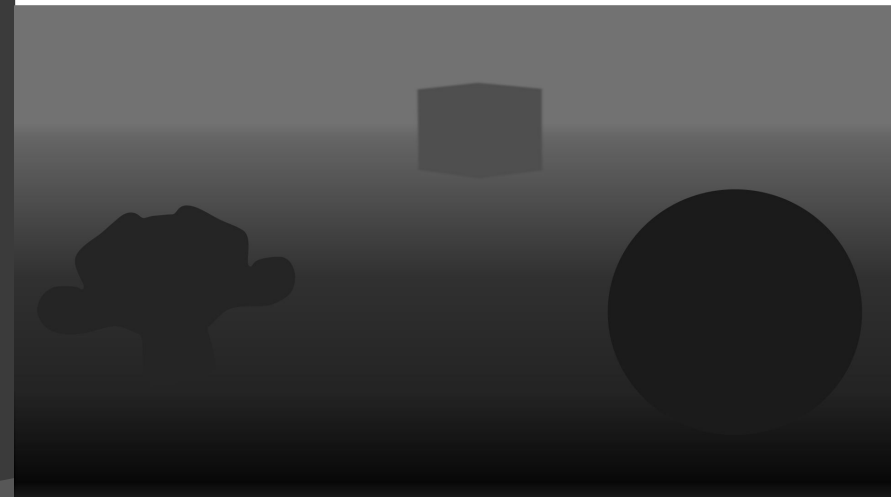


# OpenGL Buffers

- Depth buffer (z-buffer):  
stores a depth value for each pixel.
  - Depth is usually measured in terms of distance to the eye, so pixels with larger depth-buffer values are overwritten by pixels with smaller values.



A simple three-dimensional scene



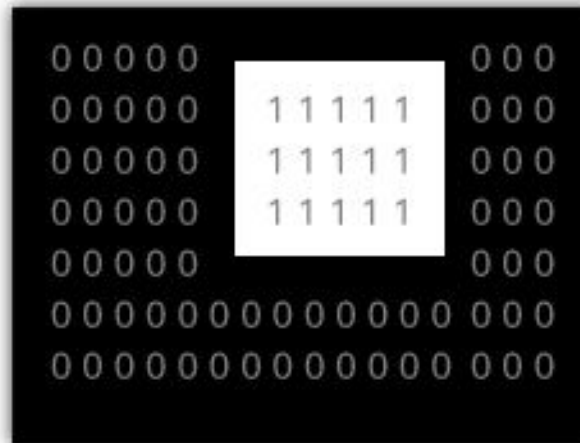
Z-buffer representation

# OpenGL Buffers

- **Stencil buffer**: One use for the stencil buffer is to restrict drawing to certain portions of the screen.
  - Other usage: real time shadows and reflections



Color buffer without stencil test



Stencil buffer



Color buffer with stencil test

# OpenGL Buffers

- Accumulation buffer: The buffer holds RGBA color data just like the color buffers do. It's typically used for accumulating a series of images into a final, composite image
- - Typical usage:
    - scene antialiasing
    - Blending
    - Motion blur

# OpenGL Buffers

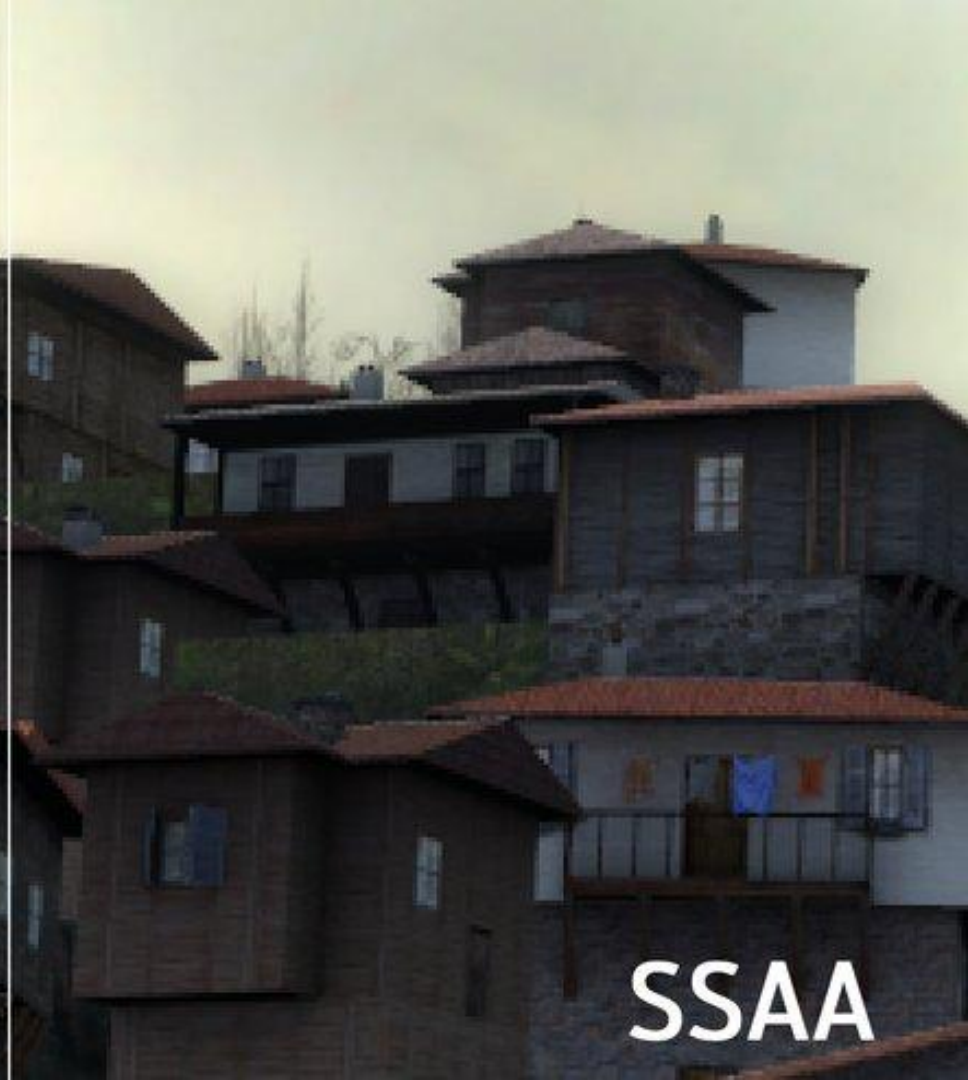
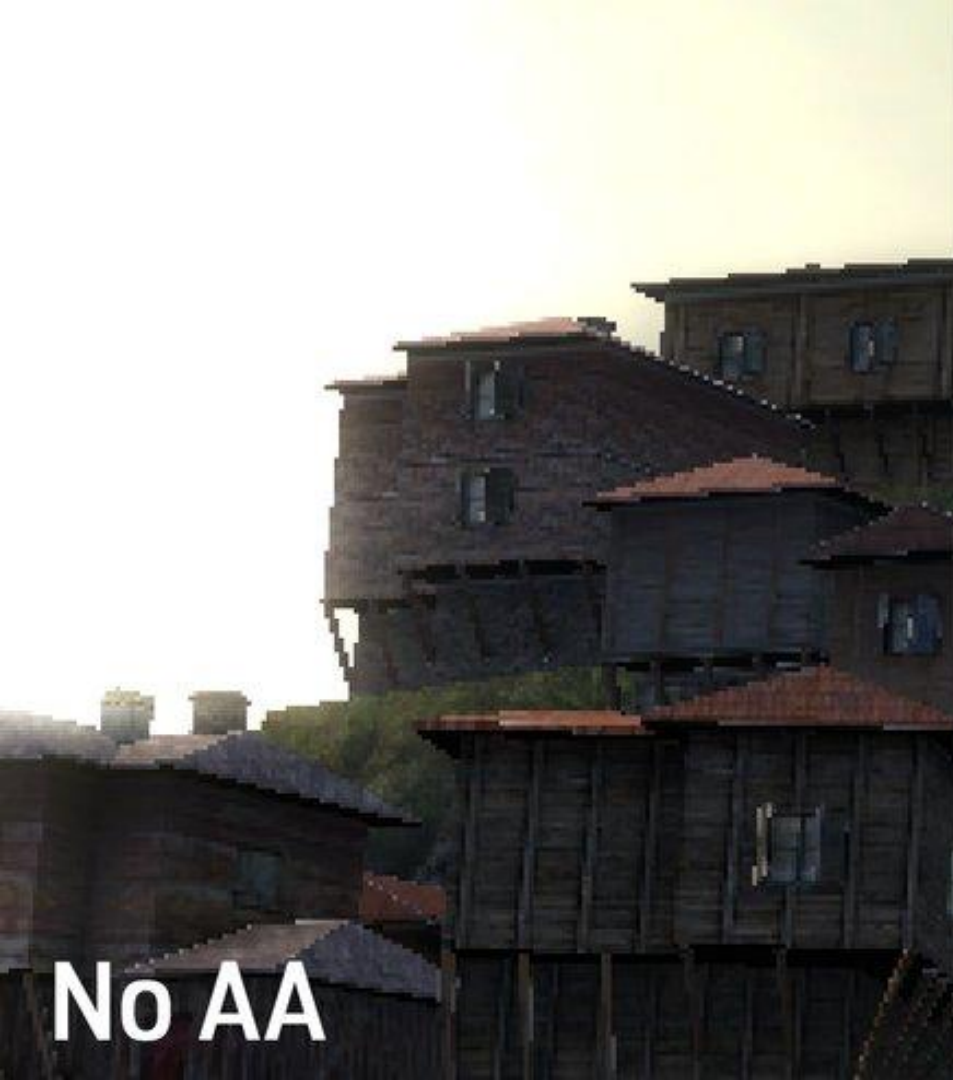
- Motion Blur Effect





# OpenGL Buffers

- Scene antialiasing



GAME OVER