VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

Lecture 11-12: View, Projections, Instancing and Introduction to Textures

CGRA 354 : Computer Graphic Programming

Instructor: Alex Doronin
Cotton Level 3, Office 330
alex.doronin@vuw.ac.nz

With slides from: Holly Rushmeier, Yale; Steve Marschner, Cornell; Taehyun Rhee, CMIC;

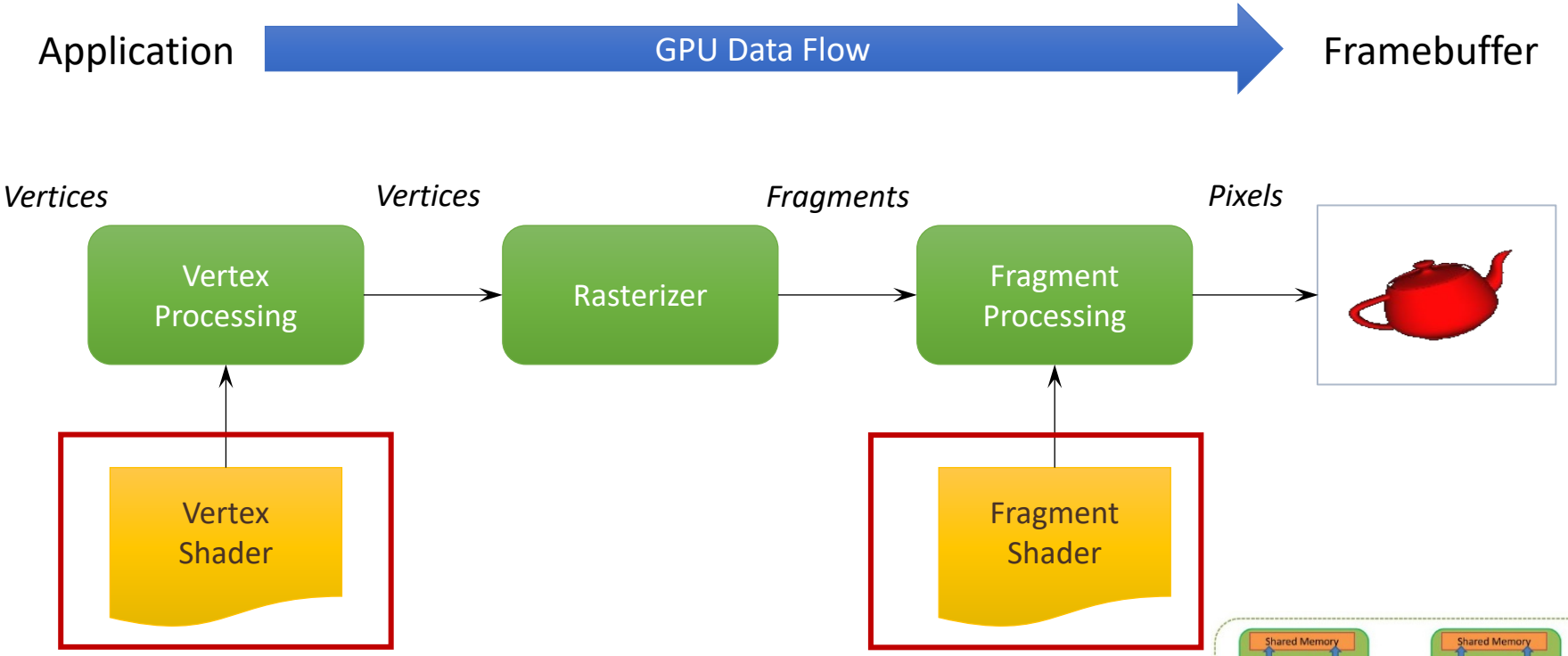
Lecture Schedule

- Lighting continued and linear algebra recap
- Transformations
- *Details of Mid-trimester test*
- View, Projection and Instancing
- *Introduction to Textures and Animation started*

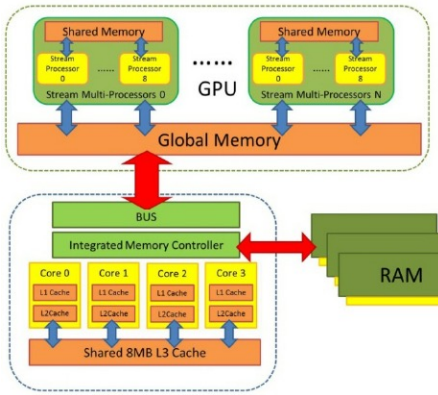
Mid-term test

- **Released:** online on Wednesday, 17th of April (Nuku), 10am
- **Duration**: 1hr (must be taken within 24h)
- **Format:** mixed (multi-choice, true/false, short answer)
- **Covers:** lecture and assignment material
- **Structure:** Core, Completion and Challenge parts

Recap: Graphics pipeline



- Modern OpenGL programs essentially do the following steps:
  - Create shader programs
  - Create buffer objects and load data into them
  - "Connect" data locations with shader variables
  - Render

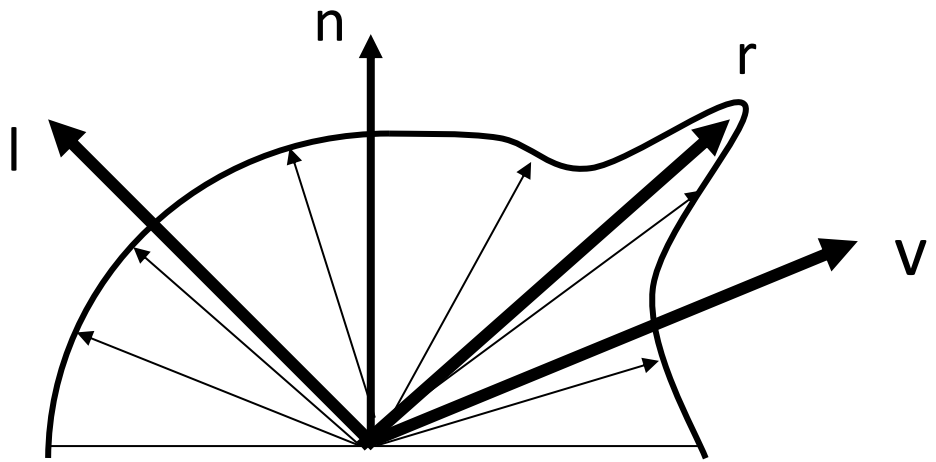


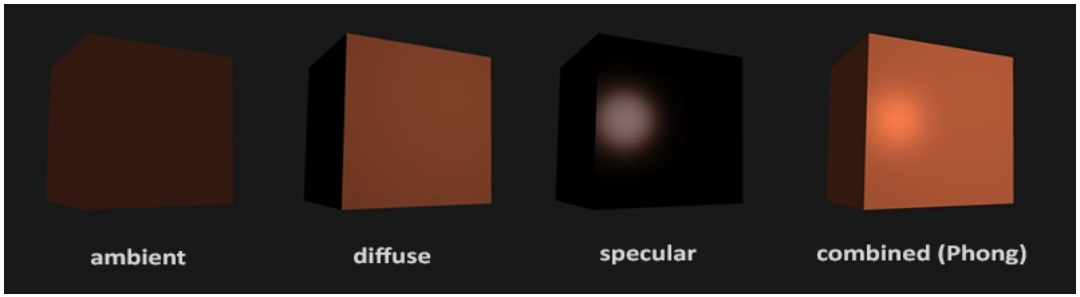
Phong Model in OpenGL

- Phong illumination model is combination of
  - Ambient i_{amb}+ Diffuse i_{diff} + Specular terms i_{sepc}
  - Developed by Bui Tuong Phong at Univ. Utah 1973

$$\mathbf{I} = k_a i_a + k_d i_d (\mathbf{n} \bullet \mathbf{l}) + k_s i_s (\mathbf{r} \bullet \mathbf{v})^{m_{shi}}$$

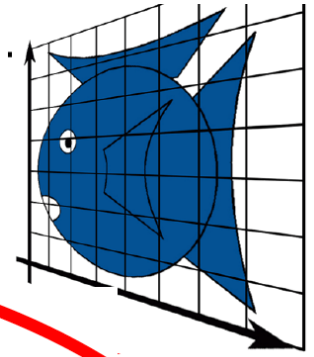
  - k_a k_d k_s are material properties having RGB components

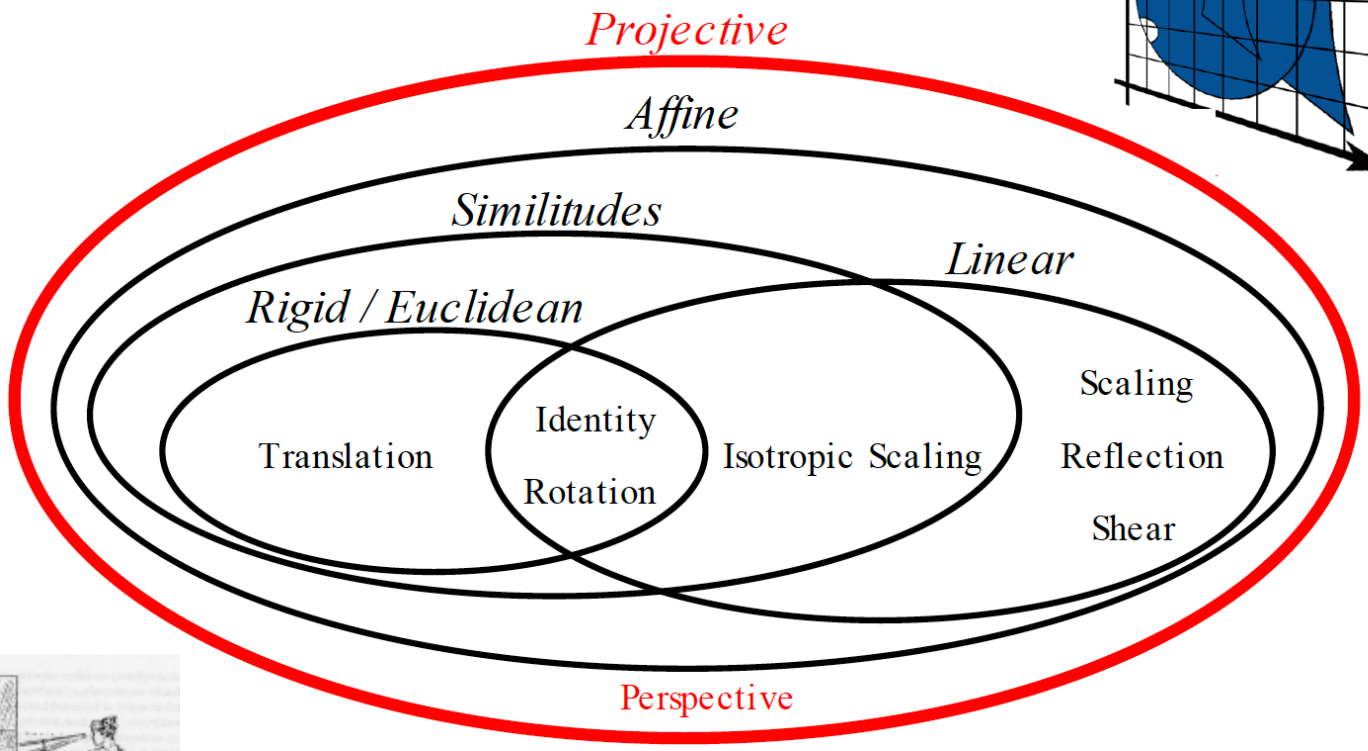


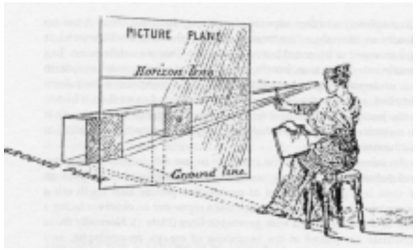


Projective Transformations

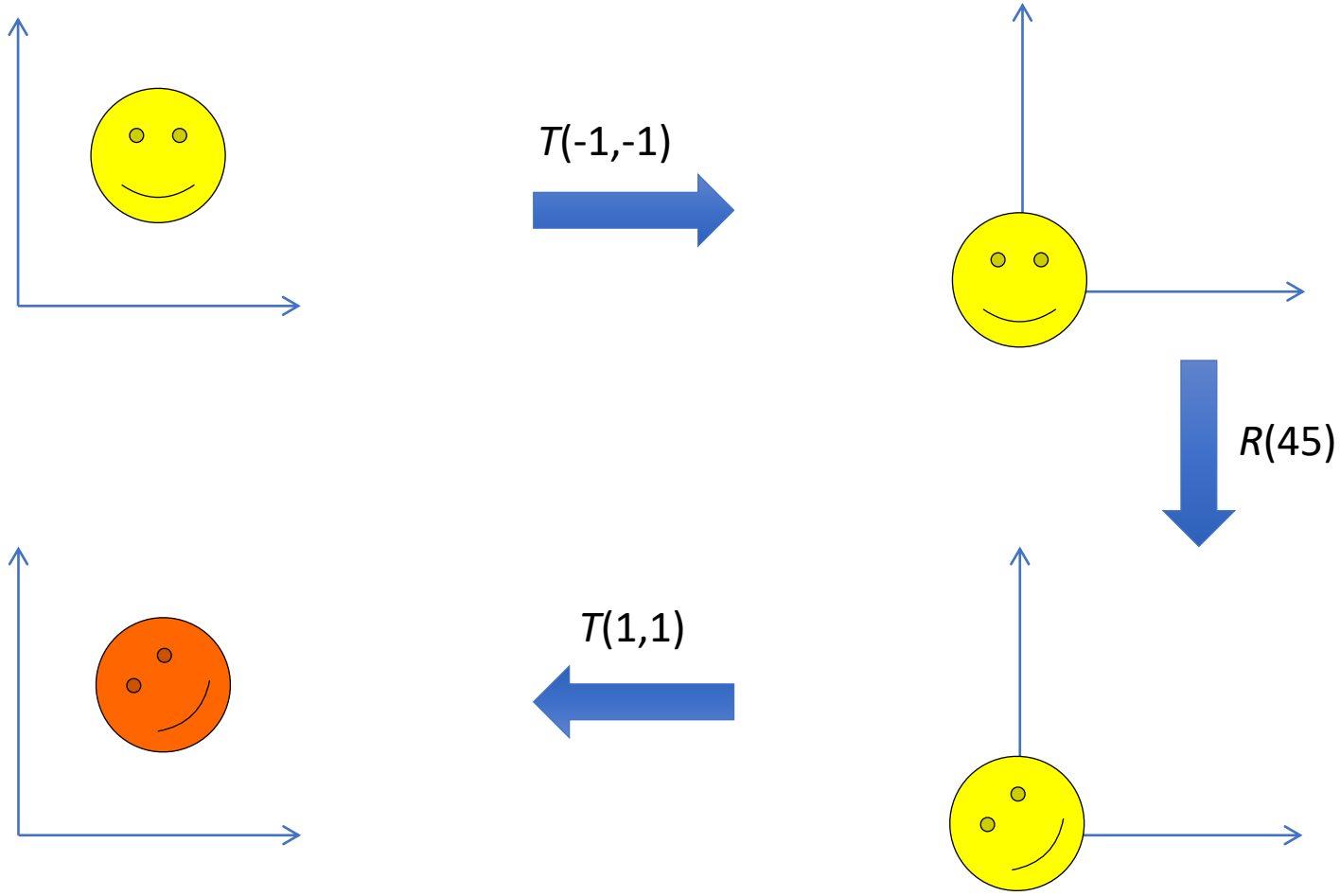
- preserves lines





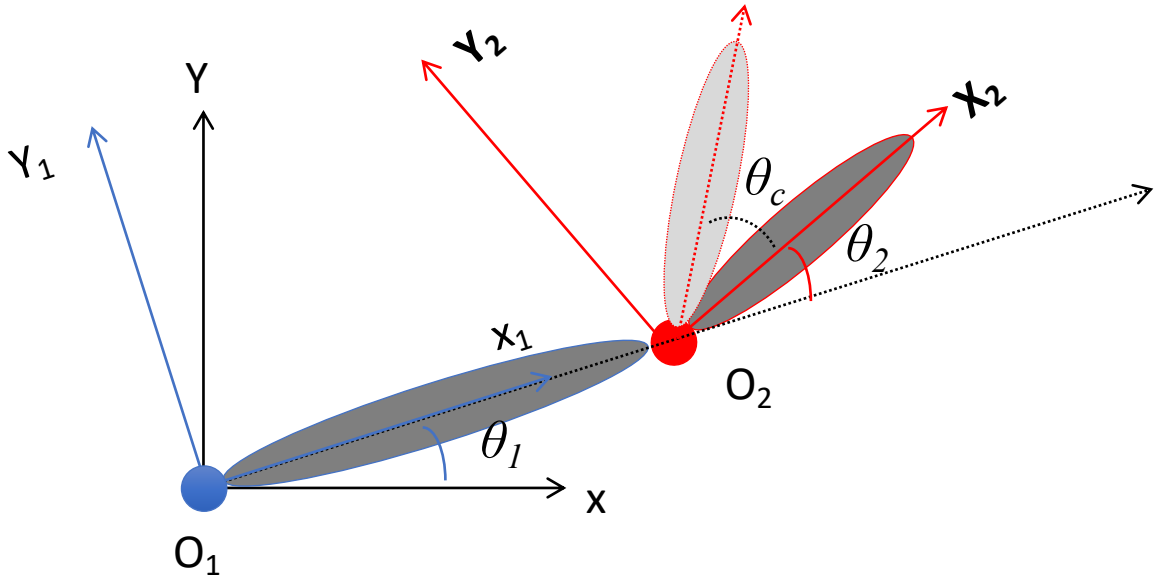


Rot(45) at (1,1)



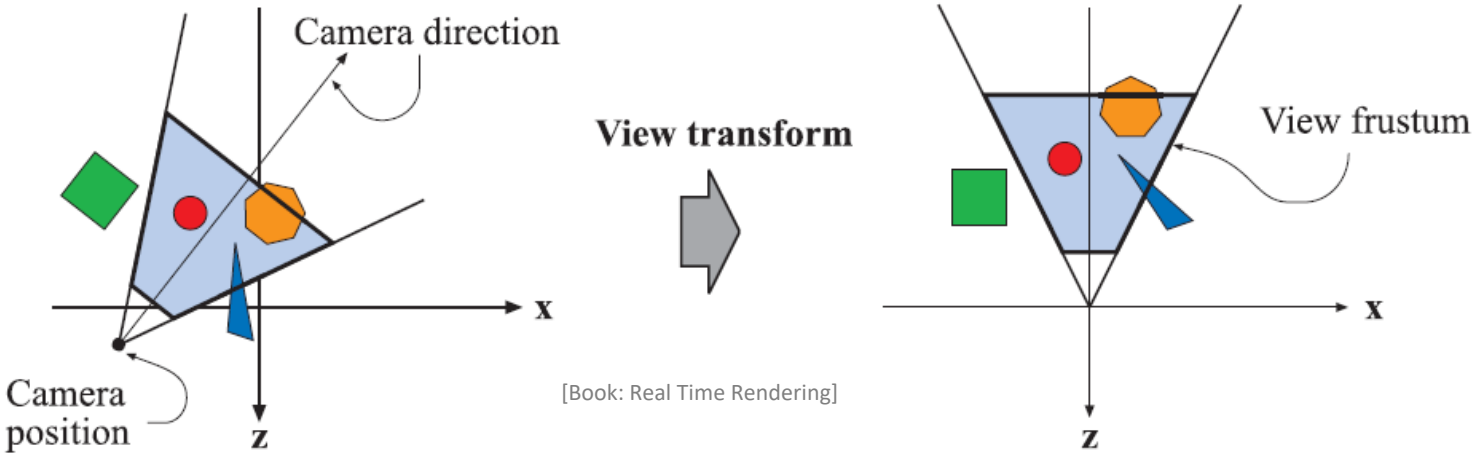
Object Coordinates

- An origin and basis define a frame of reference
- Object is defined in its local coordinates to easy control. Then, it is transferred to the world coordinates using model matrix $\mathbf{M}_{\text{model}}$



Eye(camera) coordinates

- Objects are transformed from object space to eye space using a "model" matrix
  - Combination of Model matrix $\mathbf{M}_{model}$ and View matrix $\mathbf{M}_{view}$
  - $\mathbf{M}_{model}$: from object coordinates to world coordinates
  - $\mathbf{M}_{view}$: from world coordinates to eye coordinates
  - In eye coordinates, camera is located at (0,0,0) facing –z axis



[Book: Real Time Rendering]

Move the mountains (world) or move the camera?

- Moving camera is reverse movement of objects
  - Rotate/Move Camera R_y(theta) is same as rotate object R_y(-theta)

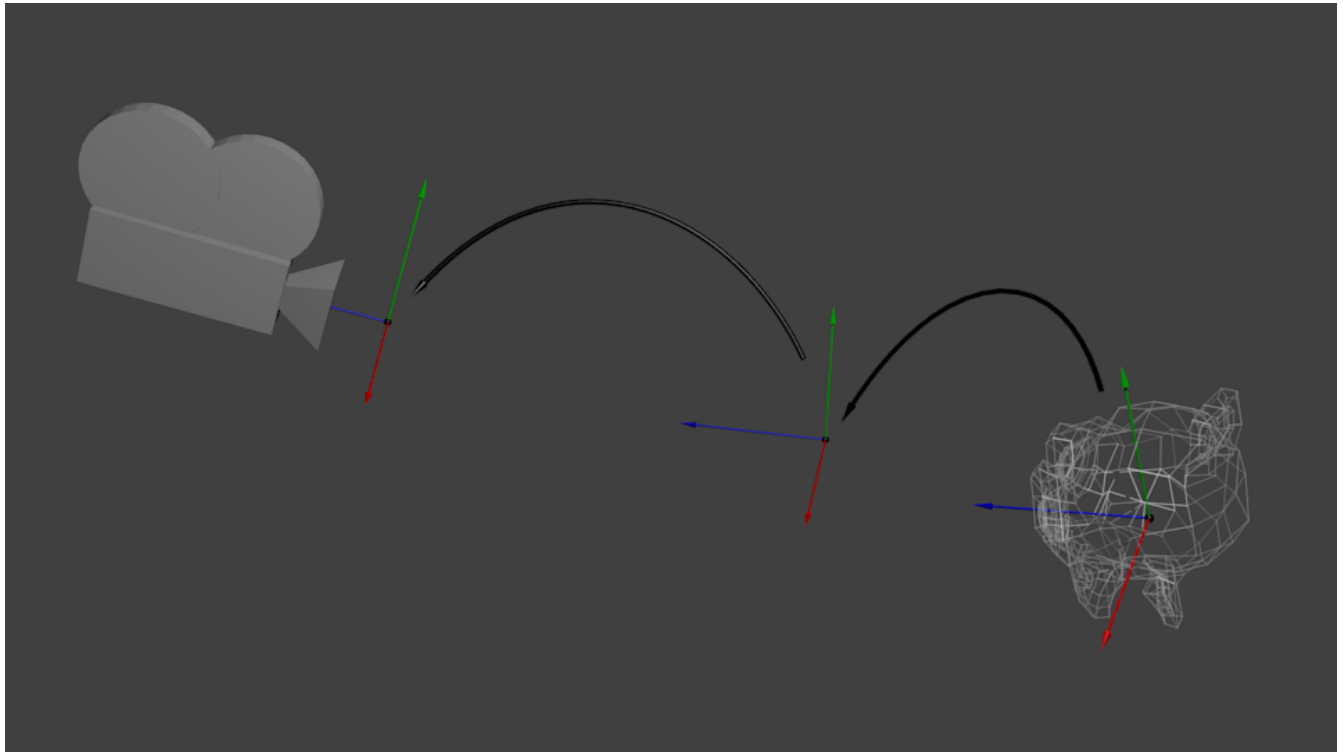


Image credit: http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

View Transformation

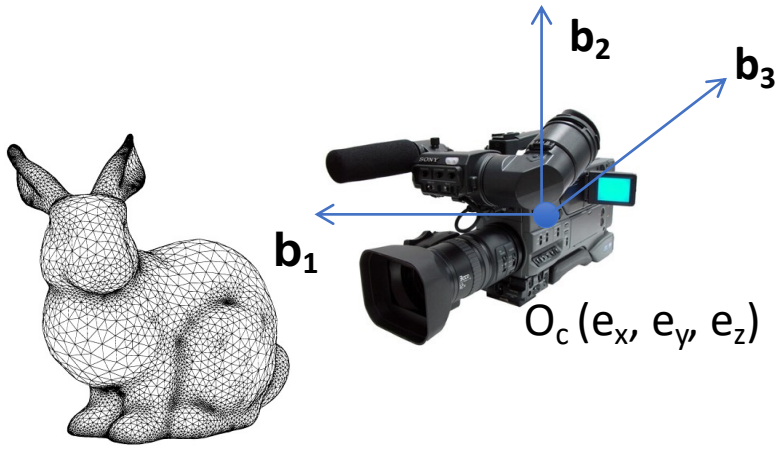
- The basis are all normalized and orthogonal
  - We can make a world coordinates transformation matrix which can move camera (position and orientation) in world coordinates
  - E.g. define a function LookAt(e_x, e_y, e_z, c_x, c_y, c_z, up_x, up_y, up_z), where

$\mathbf{b}_3 = -(\mathbf{c} - \mathbf{e})$

$\mathbf{b}_1 = \mathbf{up} \times \mathbf{b}_3$

$\mathbf{b}_2 = \mathbf{b}_3 \times \mathbf{b}_1$

$$O_c = \begin{bmatrix} b_{1x} & b_{2x} & b_{3x} & e_x \\ b_{1y} & b_{2y} & b_{3y} & e_y \\ b_{1z} & b_{2z} & b_{3z} & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} O_w$$



Parameters

`eye`	Position of the camera
`center`	Position where the camera is looking at
`up`	Normalized up vector, how the camera is oriented. Typically (0, 0, 1)

https://glm.g-truc.net/0.9.5/api/a00176.html

In the Code/Shaders

Application.cpp :

```
mat4 view = translate(mat4(1), vec3(0, -5, -m_distance)); // TODO replace view matrix with the camera transform
```

```
// display current camera parameters
ImGui::Text("Application %.3f ms/frame (%.1f FPS)", 1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
ImGui::SliderFloat("Distance", &m_distance, 0, 100, "%.1f");
ImGui::SliderFloat3("Model Color", value_ptr(m_model.color), 0, 1, "%.2f");
```

```
// cacluate the modelview transform
mat4 modelview = view * modelTransform;
```

GLM's LookAt:

```
glm::mat4 CameraMatrix = glm::lookAt(
                                     cameraPosition, // the position of your camera, in world space
                                     cameraTarget,   // where you want to look at, in world space
                                     upVector        // glm::vec3(0,1,0), but (0,-1,0) would make you looking upside-down
                                     );
```

```
for(unsigned int i = 0; i < nModels; i++)
{
    DoSomePreparations(); // bind VAO, bind textures, set uniforms etc.
    glDrawArrays(GL_TRIANGLES, 0, amount_of_vertices);
}
```

Instancing (*hint*)

- transformations allow you to define an object at one location and then place multiple instances in your scene

Instancing hint: Code/GLSL

glDrawArraysInstanced

Drawing

The function `glDrawArraysInstanced` draws multiple **instances** of the same object which allows for much greater efficiency than drawing these objects individually using calls like `glDrawArrays`. Via GLSL's built in `gl_InstanceID` or *instanced arrays* it is then possible to manipulate the vertices per instance.

The parameters of `glDrawArraysInstanced (GLenum mode, GLint first, GLsizei count, GLsizei primcount)` are as follows:

- `mode`: specifies the kind of primitive to render. Can take the following values: `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`, `GL_TRIANGLES`, `GL_QUAD_STRIP`, `GL_QUADS`, and `GL_POLYGON`.
- `first`: specifies the starting index in the enabled arrays.
- `count`: specifies the number of vertices required to render a single instance.
- `primcount`: specifies the number of instances to render.

Example usage

```
glBindVertexArray(quadVAO);
glDrawArraysInstanced(GL_TRIANGLES, 0, 6, 100);
glBindVertexArray(0);
```

Credit: https://learnopengl.com/Advanced-OpenGL/Instancing

Instancing: Code/GLSL

In the program:

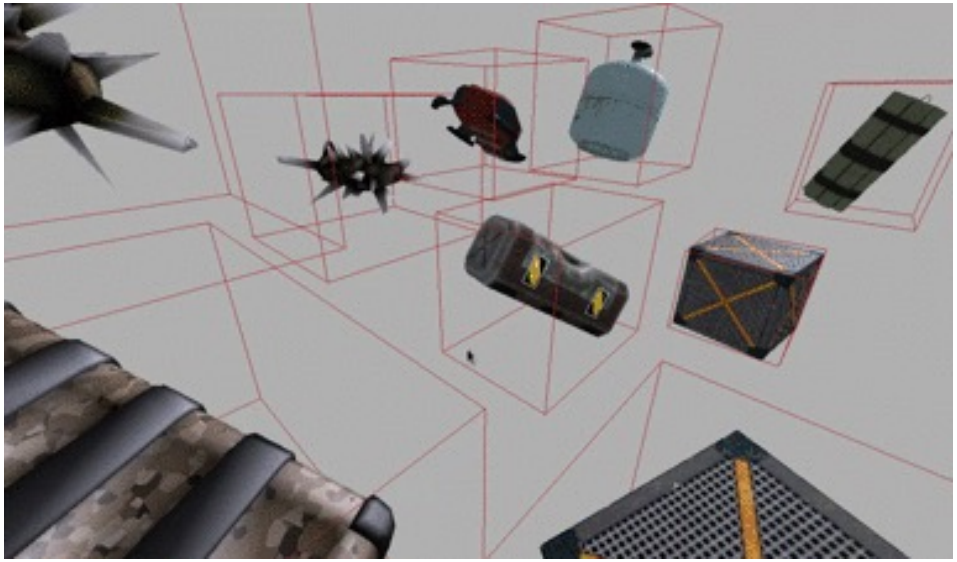
```
glm::vec2 translations[100];
int index = 0;
float offset = 0.1f;
for(int y = -10; y < 10; y += 2)
{
    for(int x = -10; x < 10; x += 2)
    {
        glm::vec2 translation;
        translation.x = (float)x / 10.0f + offset;
        translation.y = (float)y / 10.0f + offset;
        translations[index++] = translation;
    }
}
```

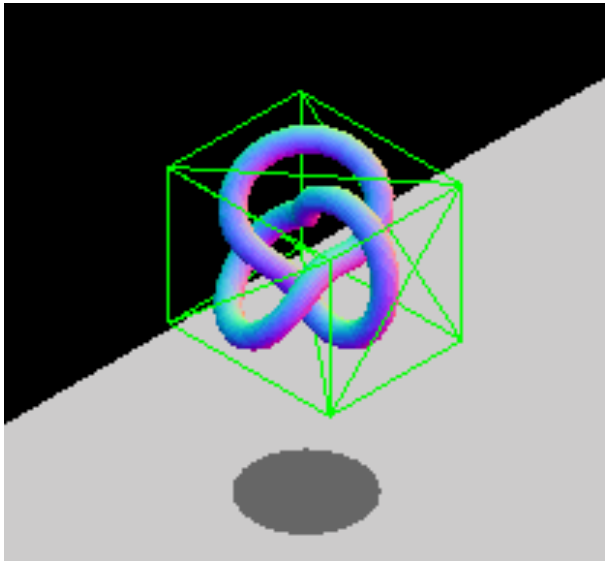
In the shader:

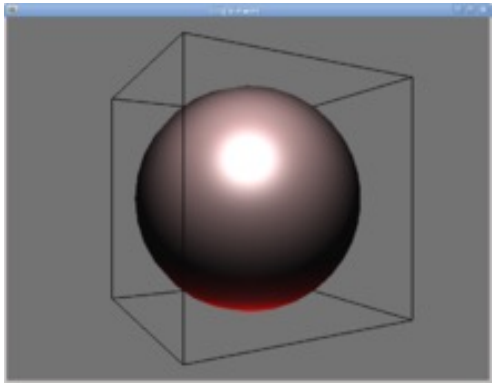
```
uniform vec2 offsets[100];

void main()
{
    vec2 offset = offsets[gl_InstanceID];
    gl_Position = vec4(aPos + offset, 0.0, 1.0);
    fColor = aColor;
}
```

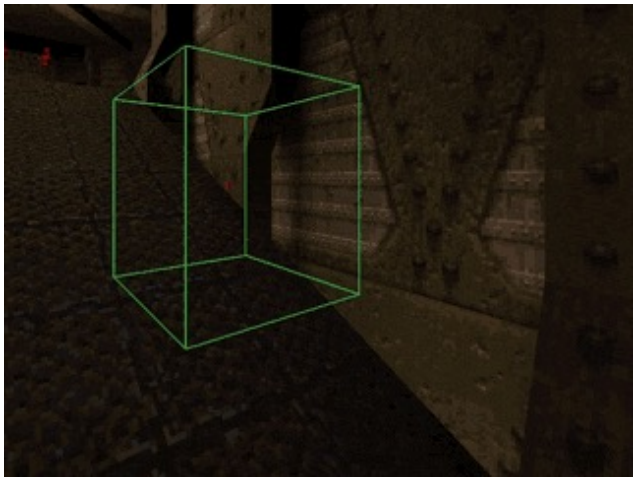
Credit: https://learnopengl.com/Advanced-OpenGL/Instancing

Object interaction: Bounding box/volume









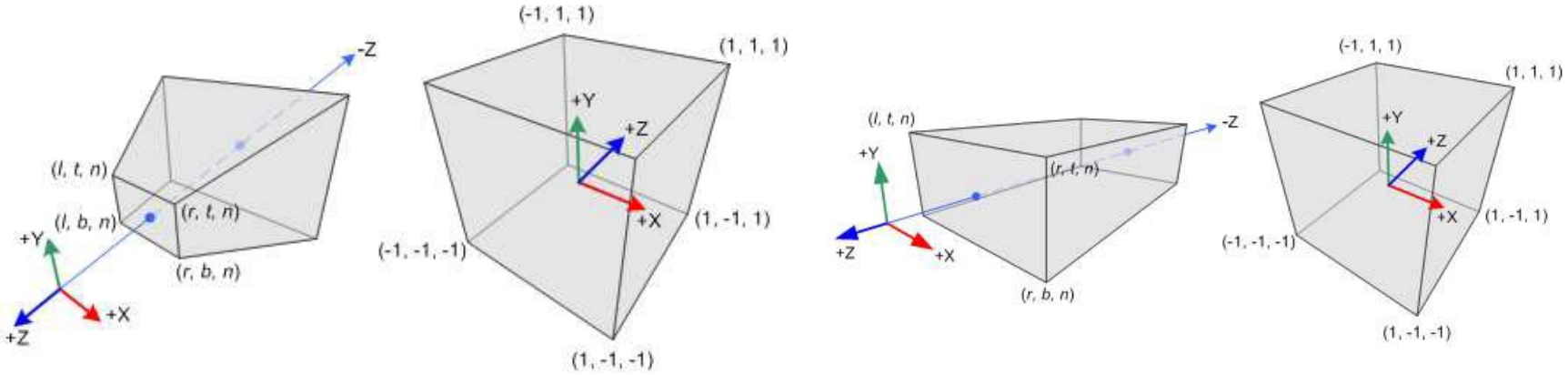
https://www.are.na/tetlie/boundingbox
https://en.wikibooks.org/wiki/OpenGL_Programming/Bounding_box

Projection

- In eye coordinates, the objects are still in 3D space
- The 3D scene in eye coordinates needs to be transferred to the 2D image on screen
- The projection matrix transfer objects in eye coordinates into clip coordinates.
- Then, perspective division (dividing with w component) of the clip coordinates
  transfer them to the normalized device coordinates (NDC)

Projection Matrix

- The projection matrix defines a view frustum determining objects to be drawn or clipped out
  - Frustum culling (clipping) is performed in the clip coordinates, before dividing points by w_c
  - Perspective Projection, Orthographic Projection

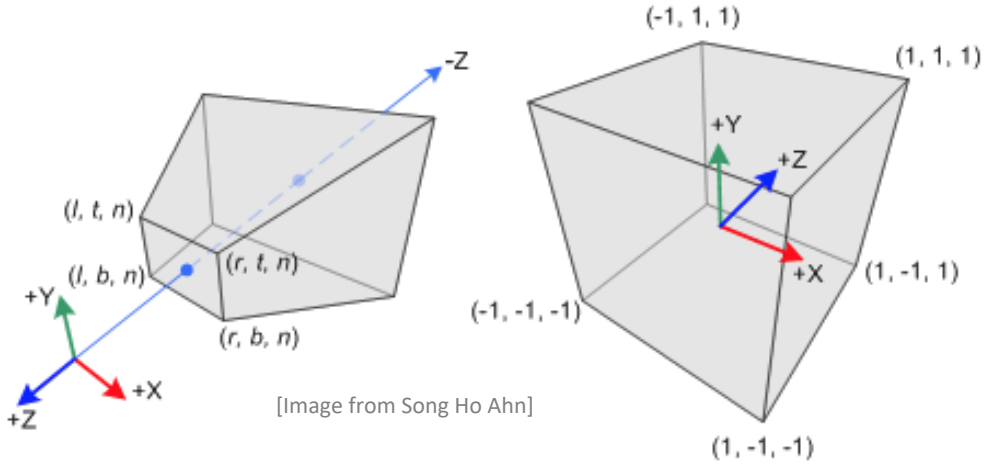


Perspective Projection [Image from Song Ho Ahn] Orthographic Projection

Perspective Projection

- 3D objects in eye coordinates are mapped into a canonical view volume
  - The view volume is specified by

    [left, right, bottom, top, near, far]
  - The view volume is transformed into a canonical view volume which is a cube from (-1,-1,-1) to (1,1,1)
    - X: [l, r] → [-1, 1]
    - Y: [b, t] → [-1, 1]
    - Z: [n, f] → [-1, 1]

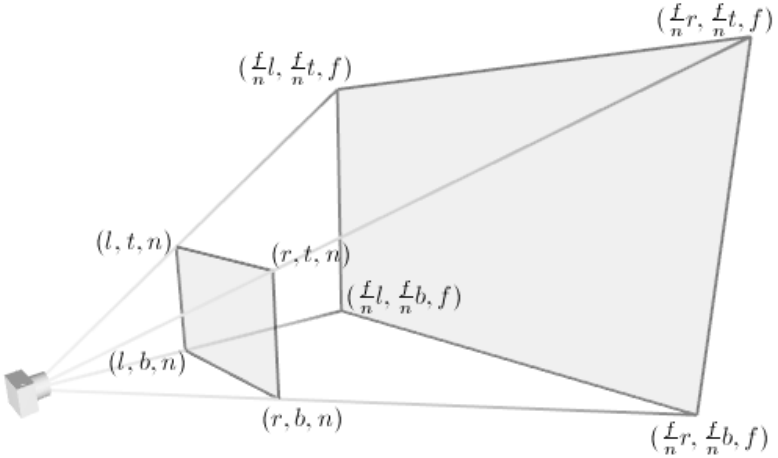


[Image from Song Ho Ahn]

Perspective Projection in OpenGL

- The perspective Projection matrix of a frustum [l ,r,b,t,n,f] is:

$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-1} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

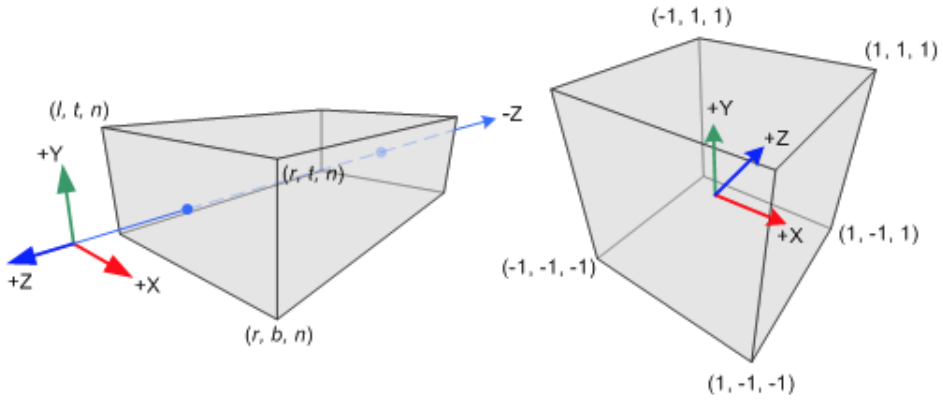


[Image from Song Ho Ahn]

```
// calculate the projection and view matrix
mat4 proj = perspective(1.f, float(width) / height, 0.1f, 1000.f);
```

Orthographic Projection

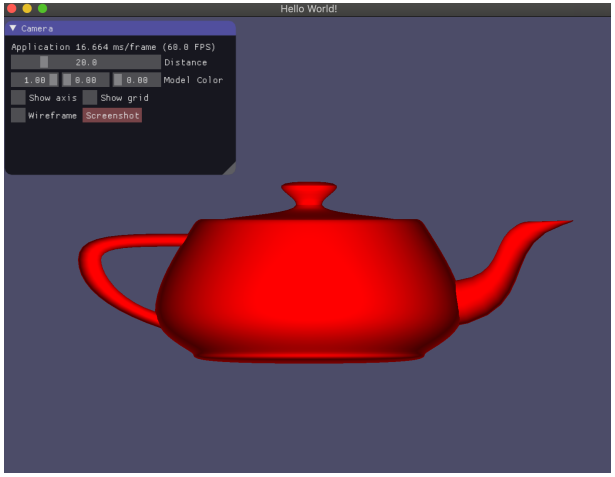
- Constructing a projection matrix for orthographic projection is much simpler
- Linear mapping from (x_e, y_e, z_e) to (x_n, y_n, z_n)



Orthographic Projection Matrix

- The Orthographic Projection matrix of [l,r,b,t,n,f] is

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-1} \\ 0 & \frac{2n}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

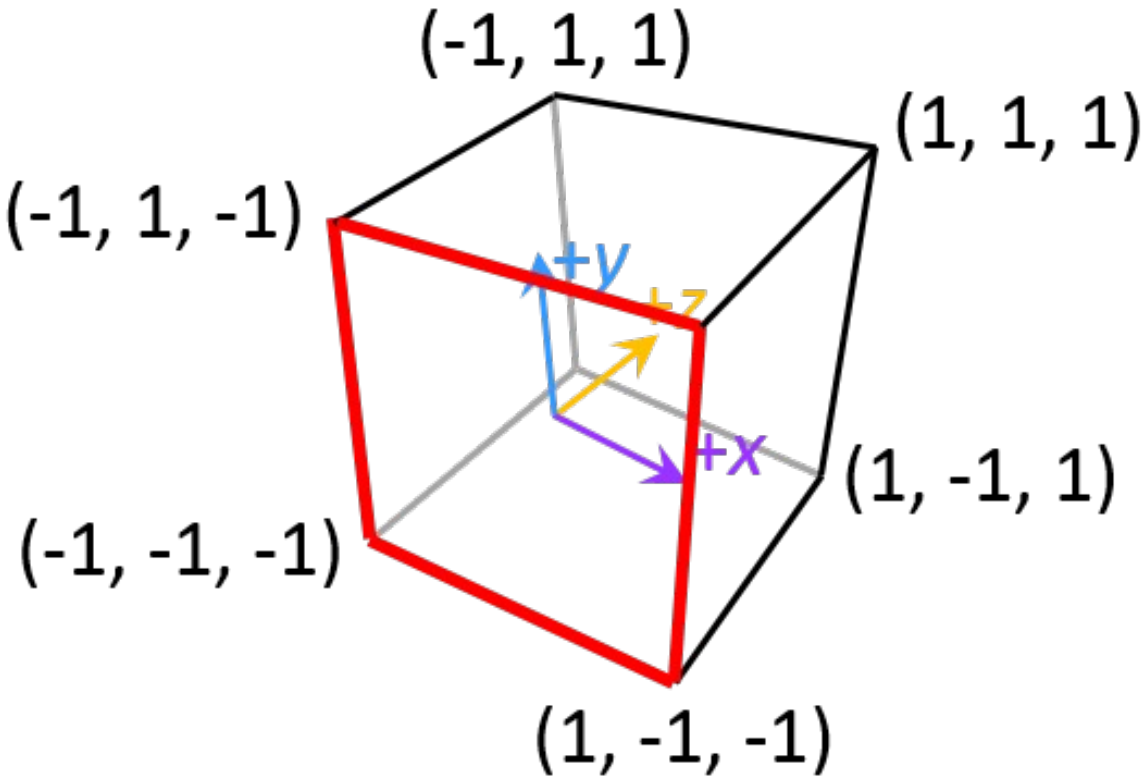


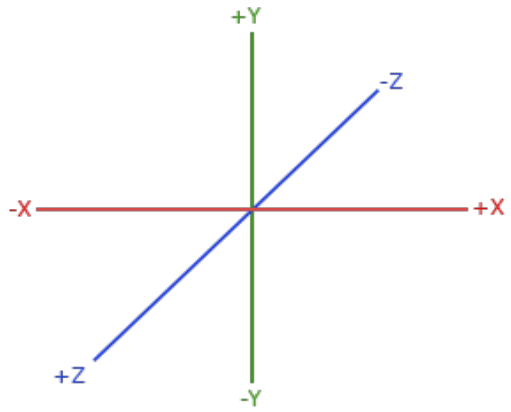
- Since W-component is not necessary, the 4th row of the matrix is remains as (0,0,0,1)

→ Try it at Home !

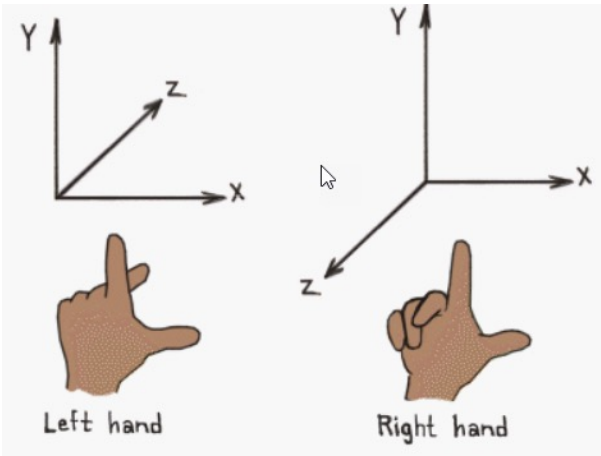
```
mat4 proj = ortho(-10.0f,10.0f,-10.0f,10.0f,0.0f,100.0f); // In world coordinates
```

Normalized Device Coordinates (NDC)





OpenGL: right-handed; others (e.g. **DirectX**: left-handed)



Normalized Device Coordinates (NDC)

- 3D Homogeneous coordinates

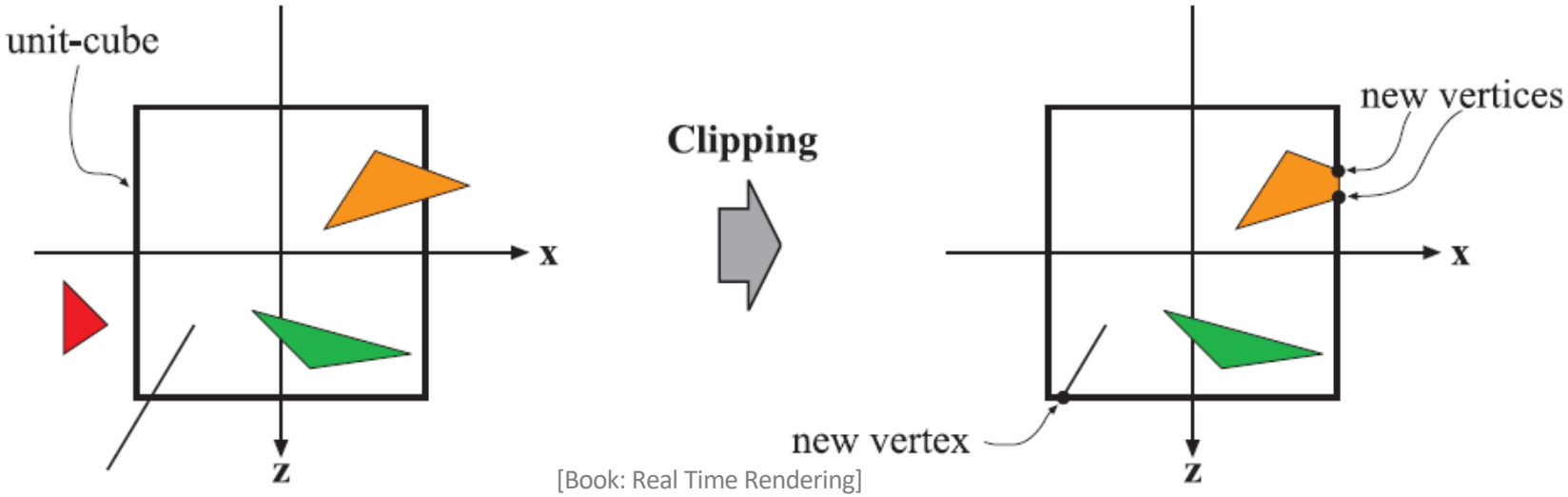
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$ can be represented as $$\begin{bmatrix} X \\ Y \\ Z \\ w \end{bmatrix}$$

where $$x = \frac{X}{w}, \quad y = \frac{Y}{w}, \quad z = \frac{Z}{w}$$

- **Normalized device coordinates (NDC)** is generated by perspective division with w

Clipping

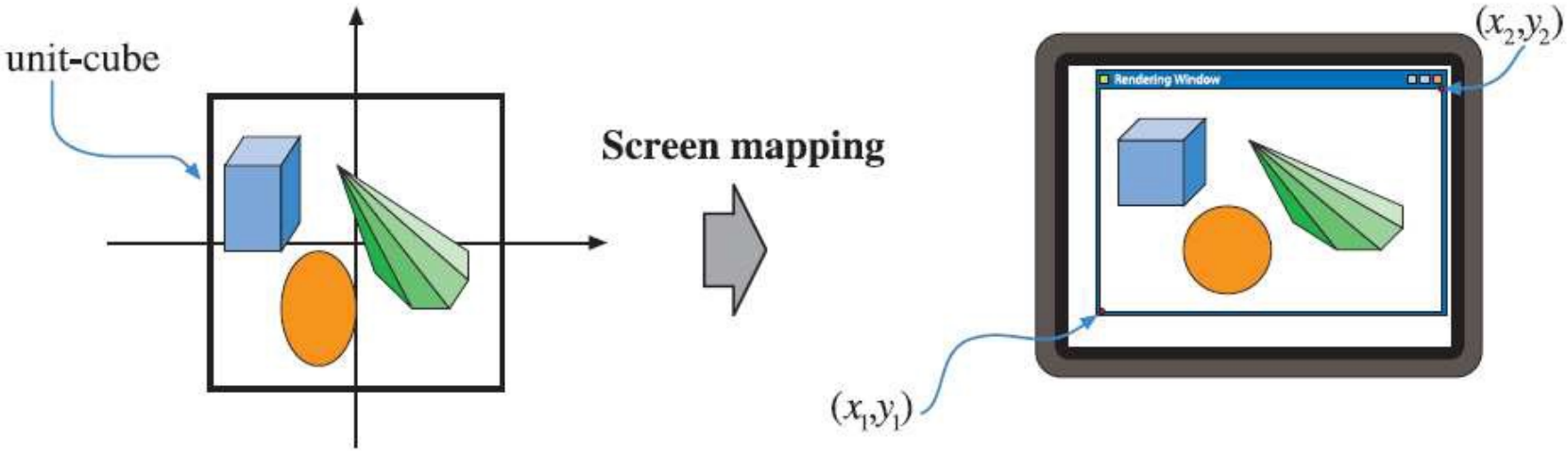
- Canonical view volume clips primitives
  - Primitives inside of the view volume are passed to the next stage
  - Primitives outside of the view volume are clipped
  - Clipping may generate new vertices



[Book: Real Time Rendering]

Viewport Transform

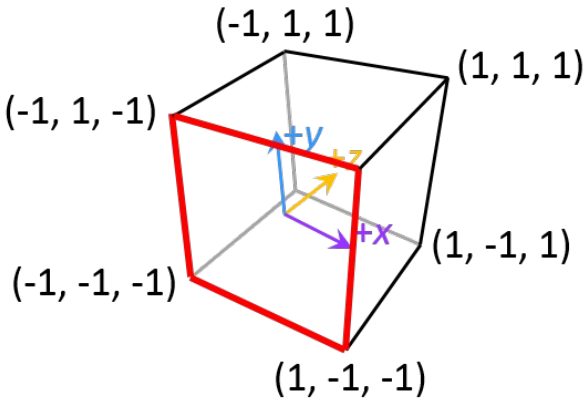
- Clipped primitives of NDC (x_n, y_n, z_n) are transferred to screen coordinates (x_s, y_s)
- Screen coordinates with depth value are window coordinates (x_w, y_w, z_w)



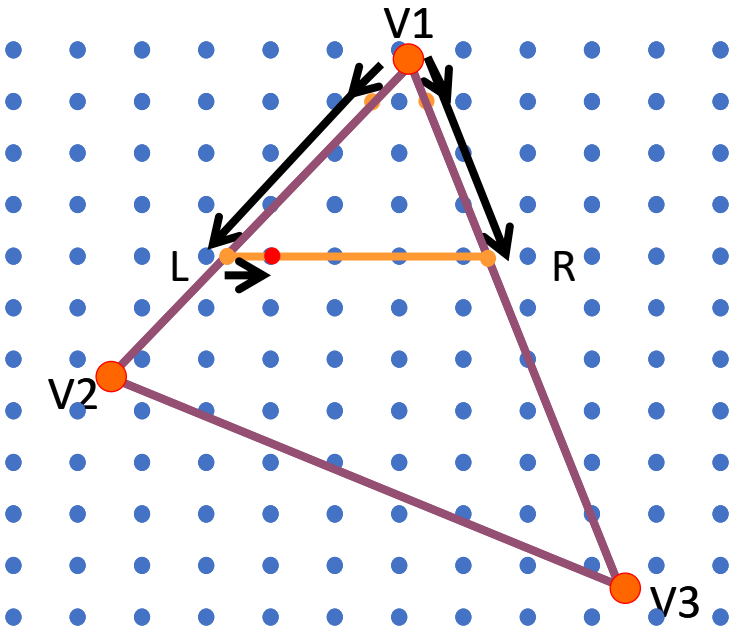
[Book: Real Time Rendering]

Viewport Transform

- (x_n, y_n) in NDC are [-1 1], the value is translated and scaled to the pixel position of screen (x_s, y_s)
- Screen coordinates (x_s, y_s) represent the pixel position of a fragment
- z_n in NDC is [-1 1], the value is translated and scaled on [0 1] for z_w
- z_w is the depth value of the pixel position (x_s, y_s) used for depth test using z-buffering



Rasterization



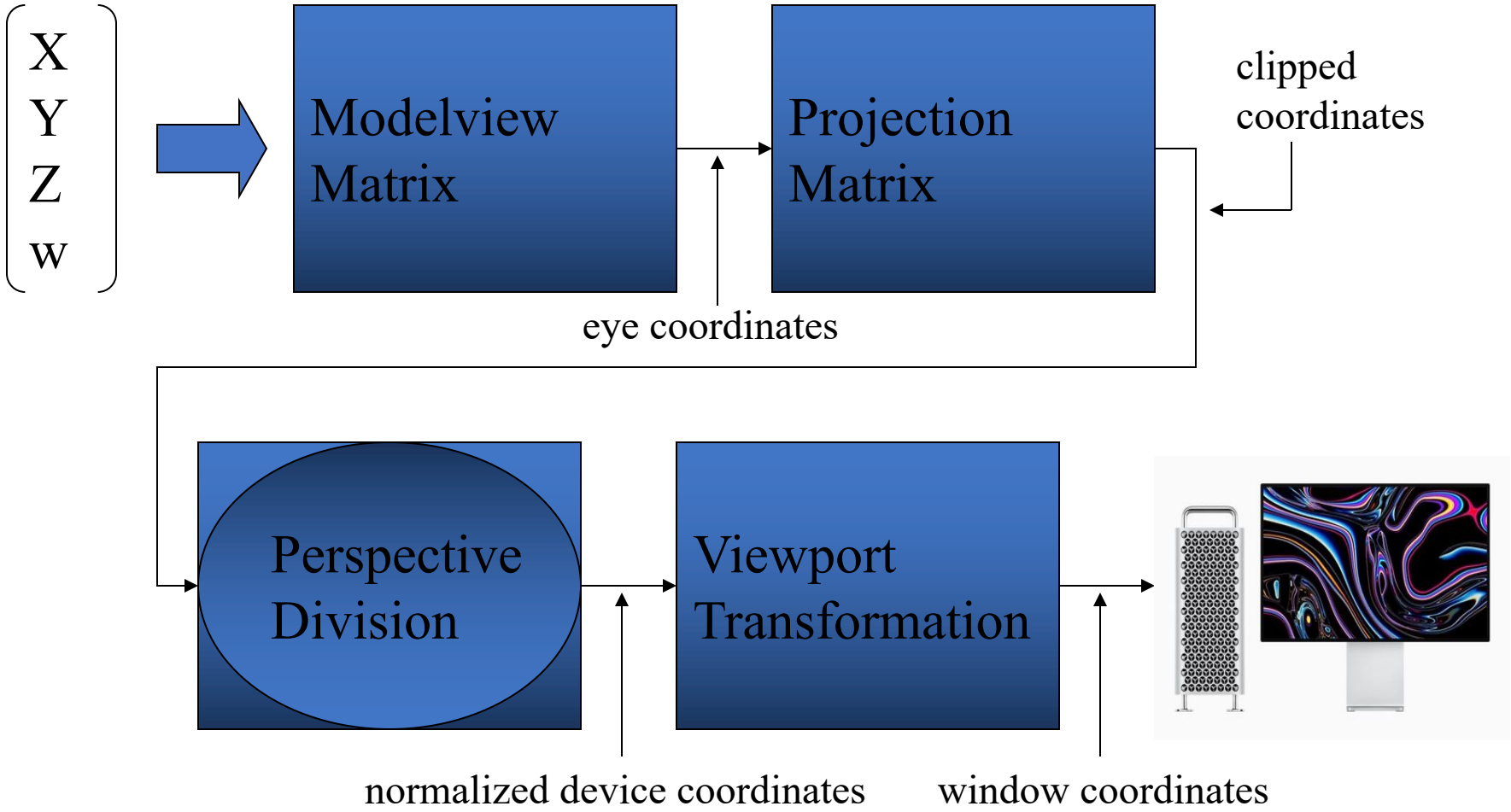
- Convert primitives into fragment
  - Interpolates triangle vertices into fragments
  - Fragments are mapped into frame buffer

Hidden Surface Removal

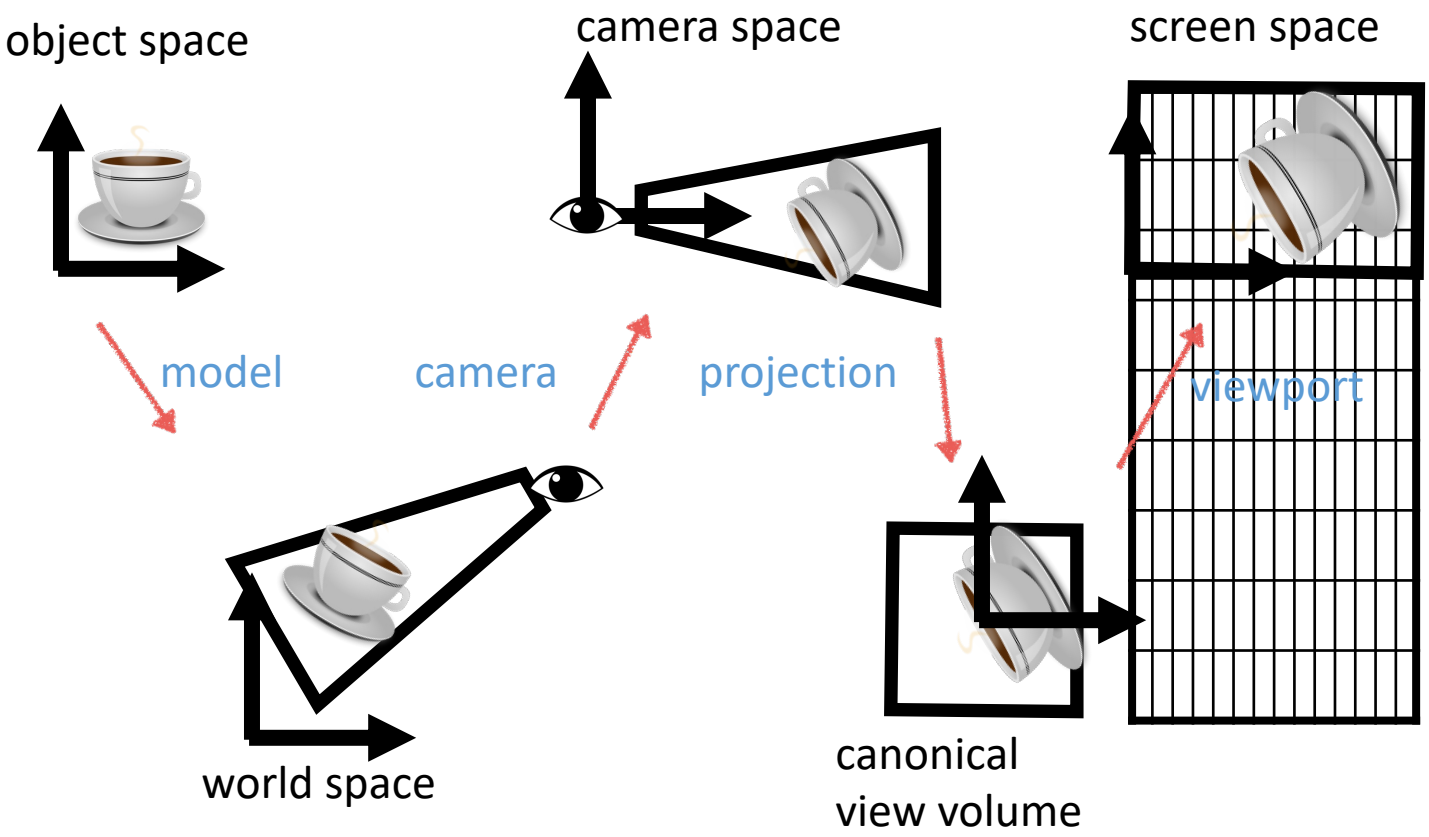
- Eliminate parts that are occluded by others
  - Depth buffer (z-buffer) contains the nearest depth values of each fragment
  - If (current depth < depth buffer),
    update frame buffer and depth buffer
    using the current values (color/depth)

```
glEnable(GL_DEPTH_TEST);
    ...
while (1)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    draw_3d_object_A();
    draw_3d_object_B();
}
```

Stages of Vertex Transformations



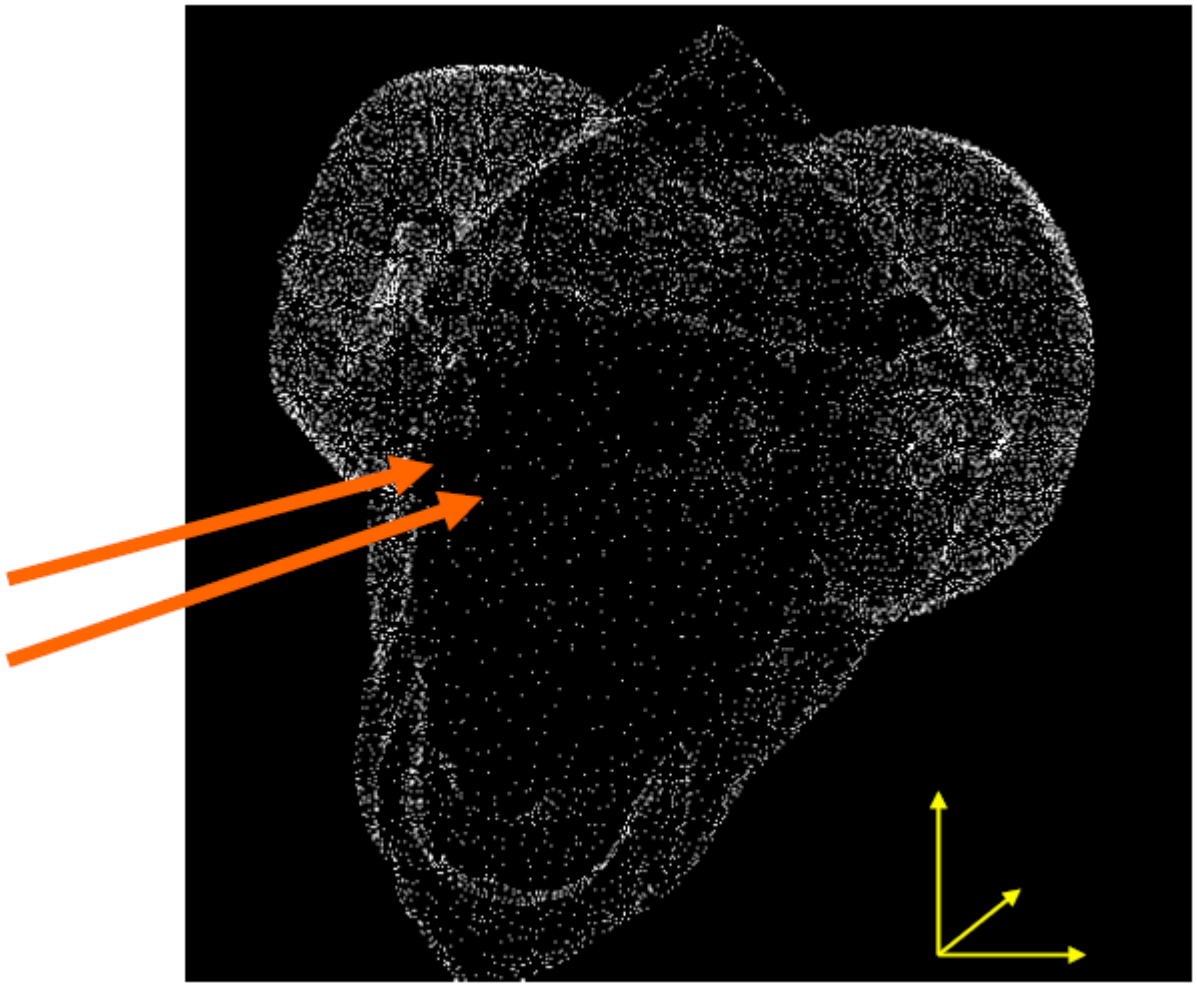
Viewing Transformation



Surface Details

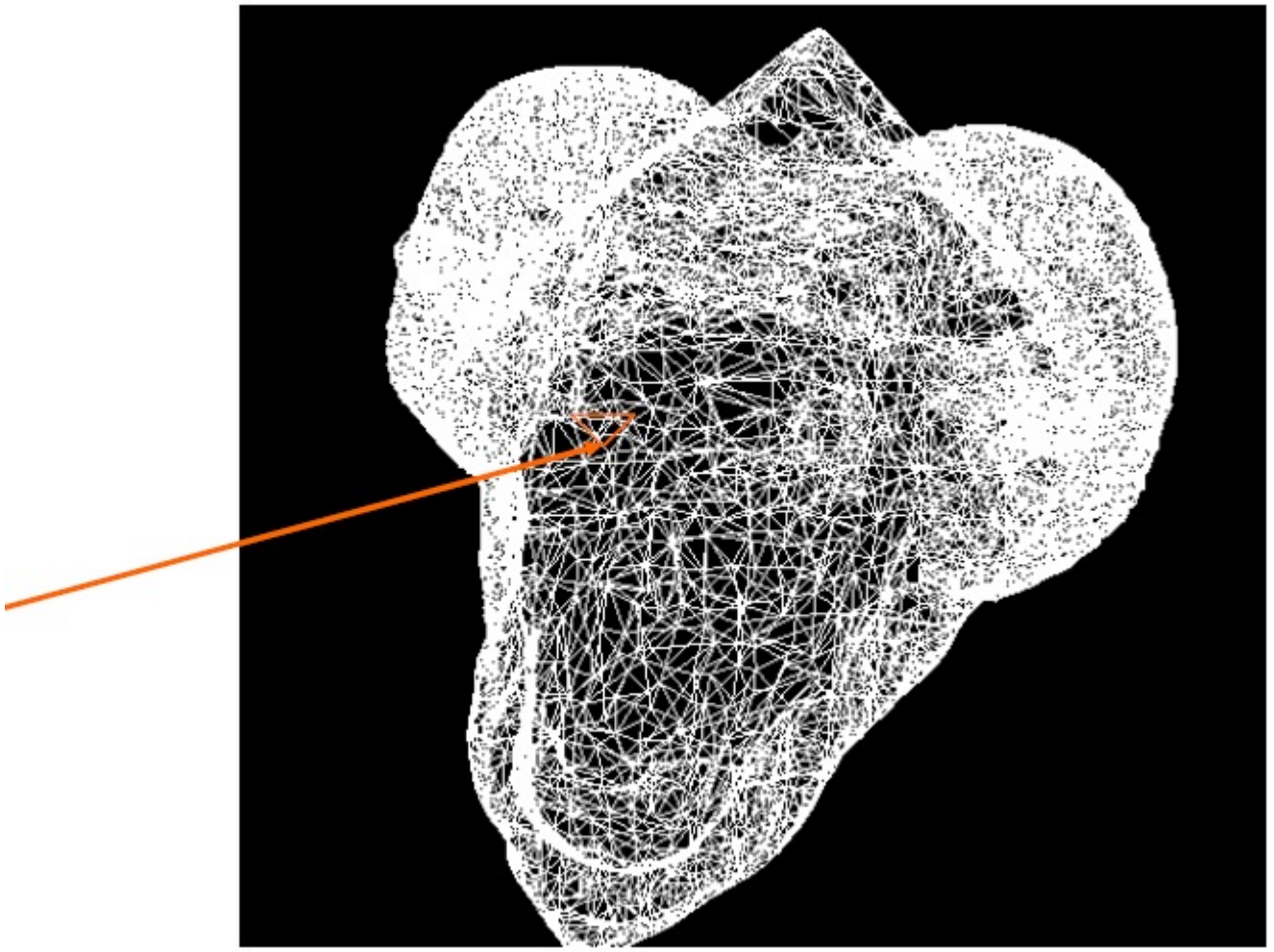
Digital Representation of Objects

The triangles are defined to connect a set of points



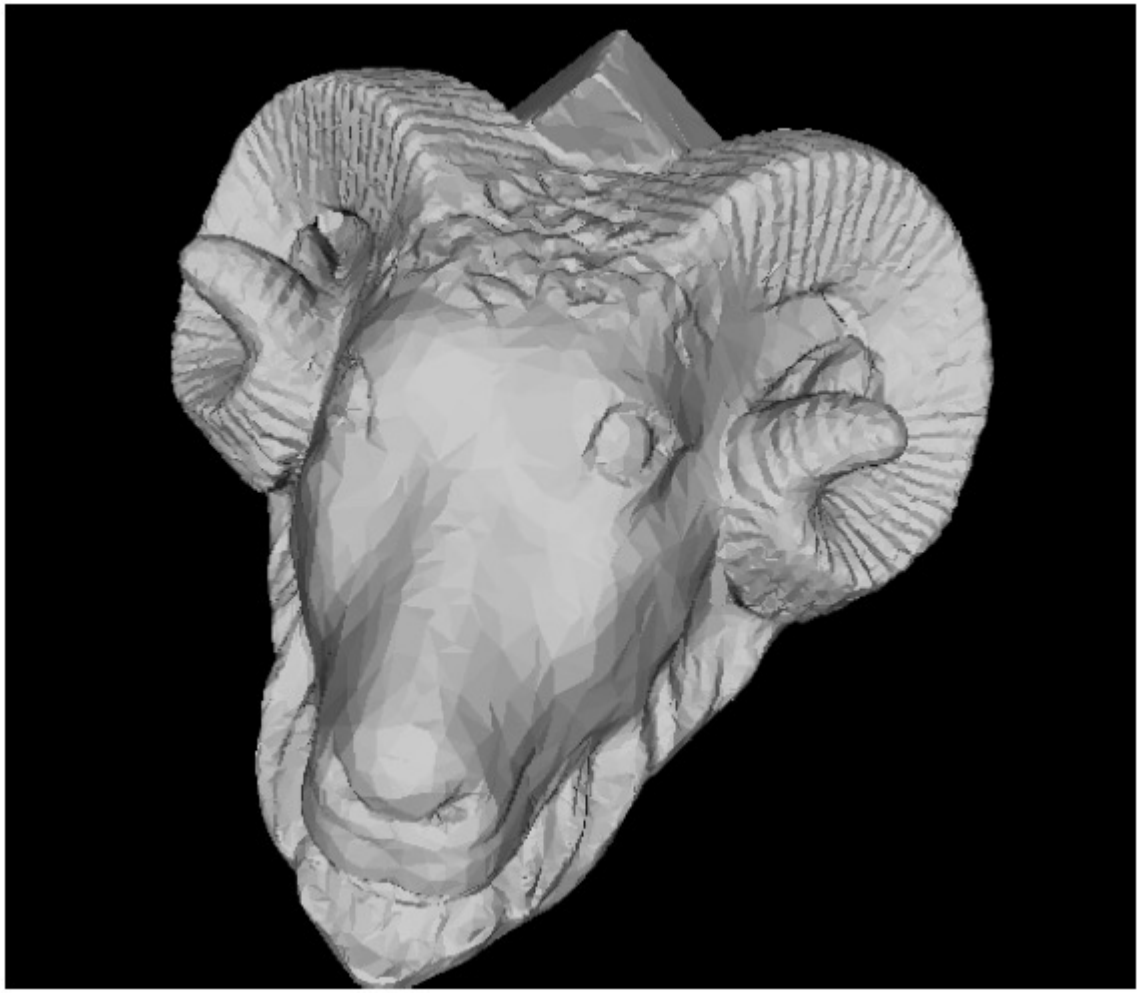
Digital Representation of Objects

The points are connected by edges.

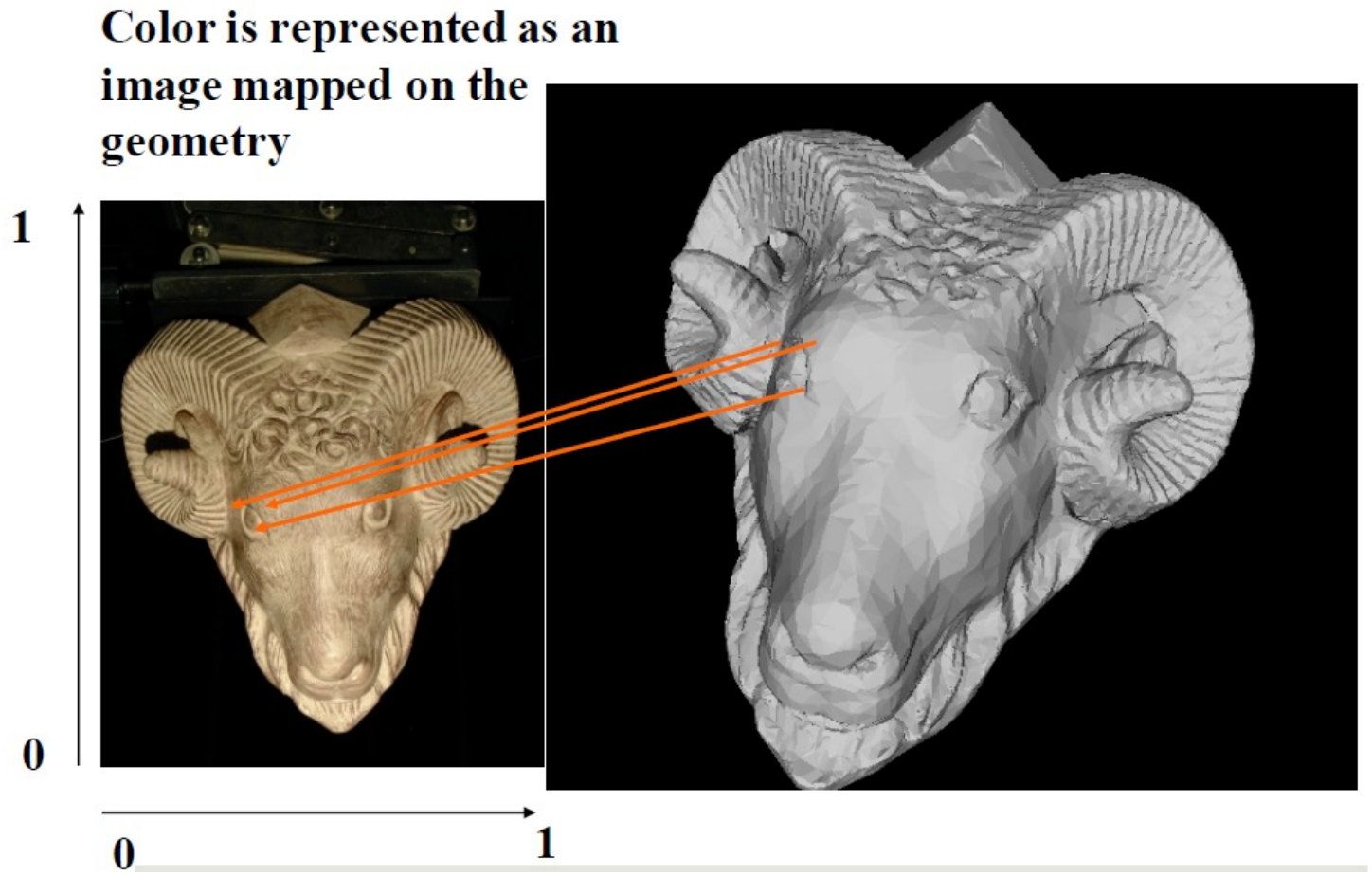


Digital Representation of Objects

The shape is defined by the set, or mesh, of triangles that are defined by the edges.



Digital Representation of Objects

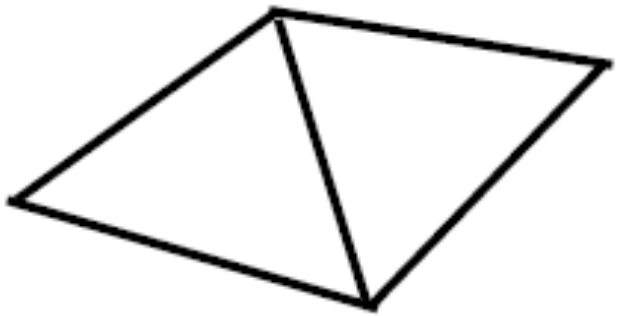


Digital Representation of Objects

Different versions of the model may have different numbers of points and triangles, many images may be mapped to the surface.

Why images, and not data/vertex?

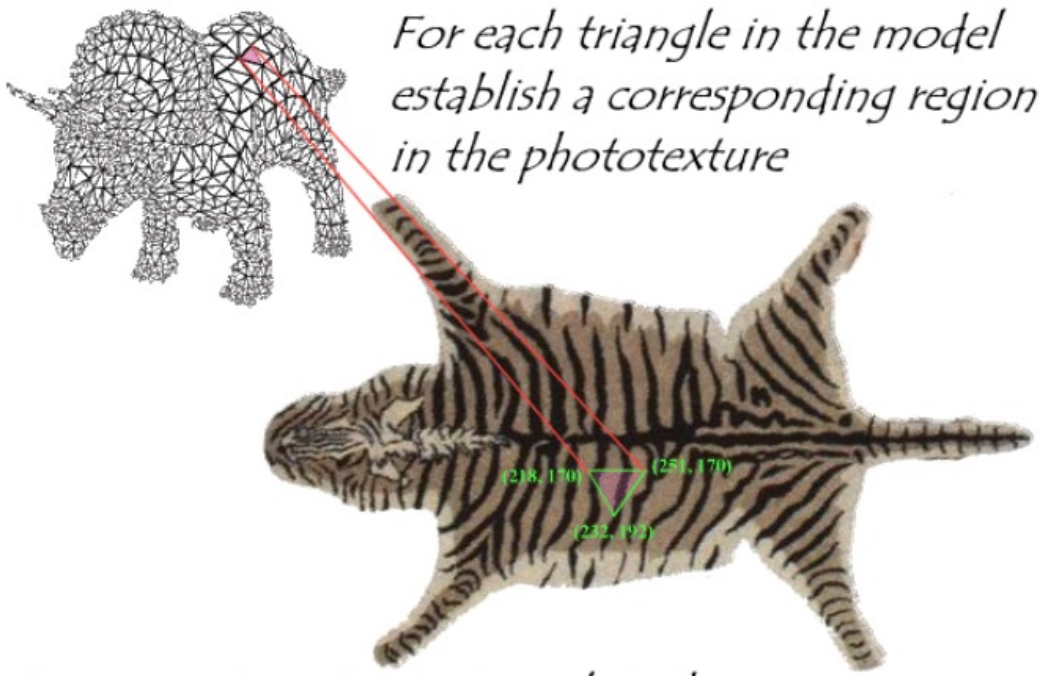
Triangles: need to store coordinates and connectivity



In an image coordinates and connectivity are implied by order

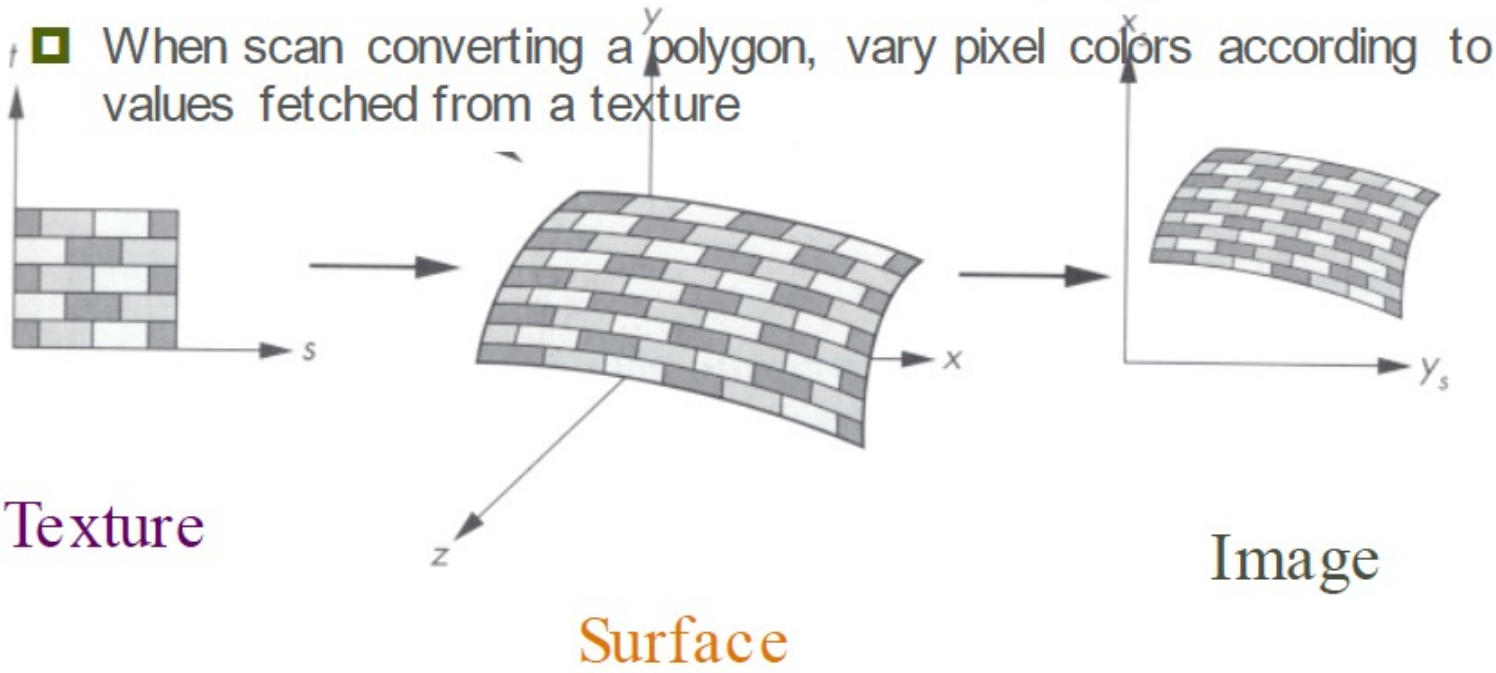
Photo-textures

The concept is very simple!



Textures

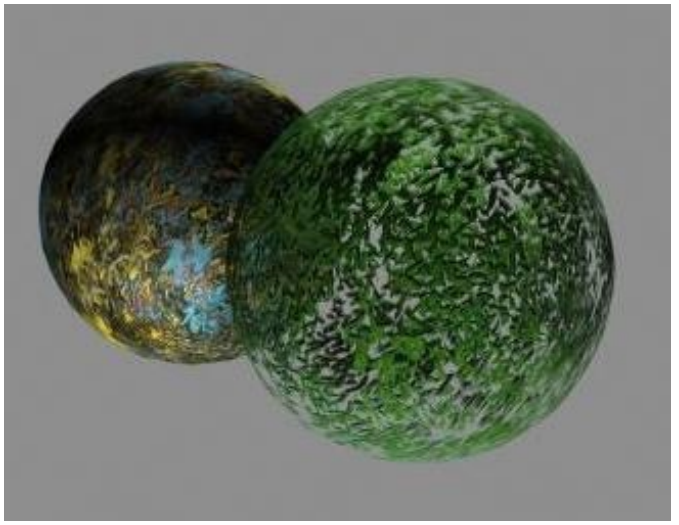
- Describe color variation in interior of 3D polygon
  - When scan converting a polygon, vary pixel colors according to values fetched from a texture



Texture Mapping

- Texture mapping is a method for adding detail using surface texture (a bitmap or raster image), or color to a computer-generated graphic or 3D model [wikipedia]
- Developed by Dr. Edwin Catmull (Ph.D. thesis ’74)

http://davidevans3d.wordpress.com/2013/03/29/teapot-texture-2/



http://www.siggraph.org/education/materials/HyperGraph/mapping/texture2.htm

Why Texture mapping ?

- Add more details and realisms without increasing geometric complexity

[Virtual fighter 1993 SEGA]

[Street fighter X Tekken 2010 Capcom]

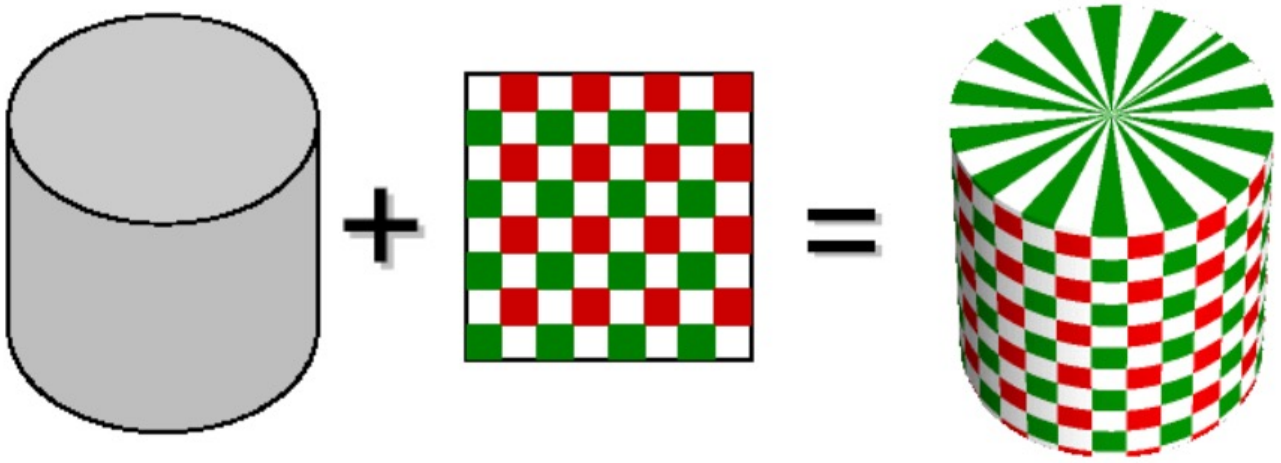
Why Texture mapping ?

- Add more details and realisms without increasing geometric complexity
  - Save efforts, memory, and computation cost for creating geometry and rendering
  - Add details per fragment
    - Color
    - Material
    - Geometry (normal)
  - Add visual effects for real-time rendering
    - Shadow, light, reflection

Texture Mapping Issues

- Texture mapping methods
  - Parameterization
  - Mapping
  - Filtering
- Texture mapping applications
  - Basic textures
  - Modulation textures
  - Illumination mapping
  - Bump mapping
  - Displacement mapping

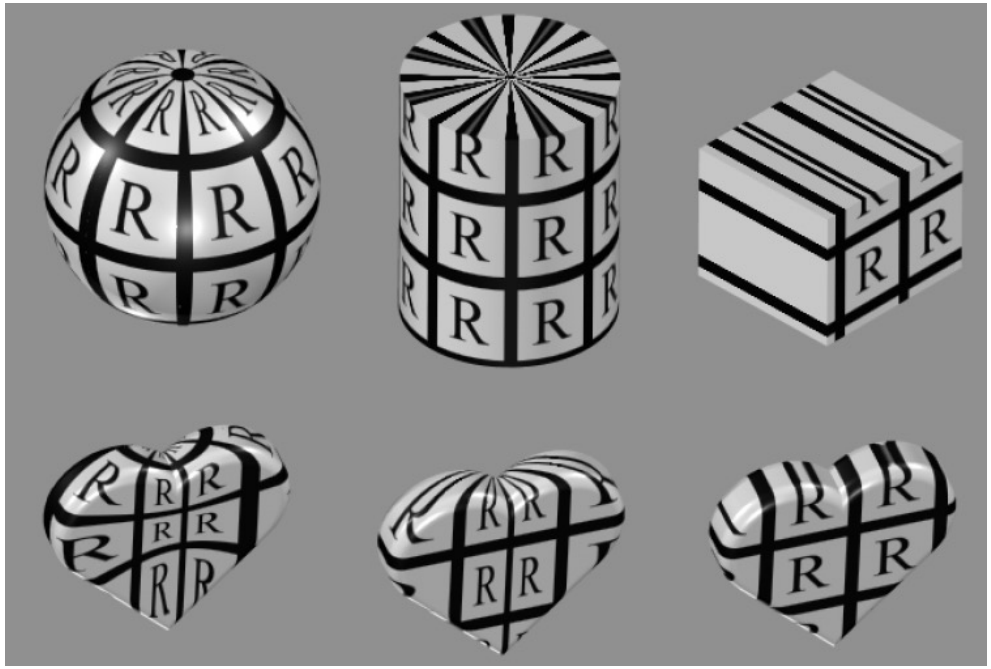
Parameterization



- Q: How do we decide *where* on the geometry each color from the image should go?

Vary the Projection

- Generate texture coordinates using surface parameterization



Spherical Projection

Cylindrical Projection

Planar Projection

Overview of Texture mapping

- Textures are represented as 2D/3D images
  - It can store a various properties such as colors, normals, environment, lighting, etc
  - The basic element of the texture is ‘texel’
- Textures are mapped with the geometry using the texture coordinates (u, v)
  - Texture coordinates are assigned to a vertex
    - Multiple texture coordinates for multi-texturing
  - Mapping from object space to texture space
  - Some times need 3D or 4D (s, t, r, q)

Texture mapping process

- Texture mapping is applied per fragment
  - Rasterization determines fragment positions and interpolates texture coordinates from adjacent vertices
  - Texture lookup is performed per fragment using interpolated texture coordinates

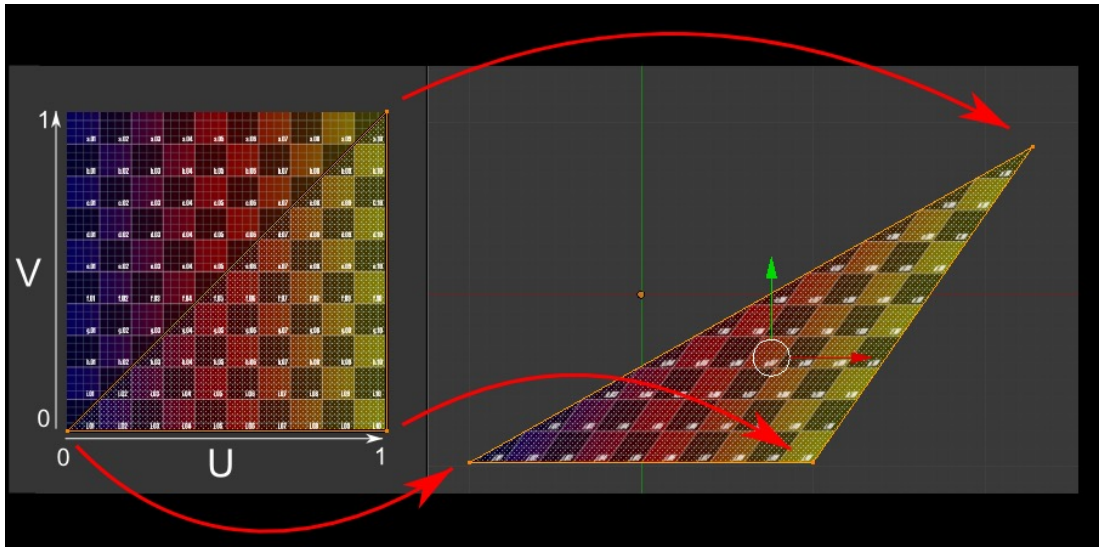


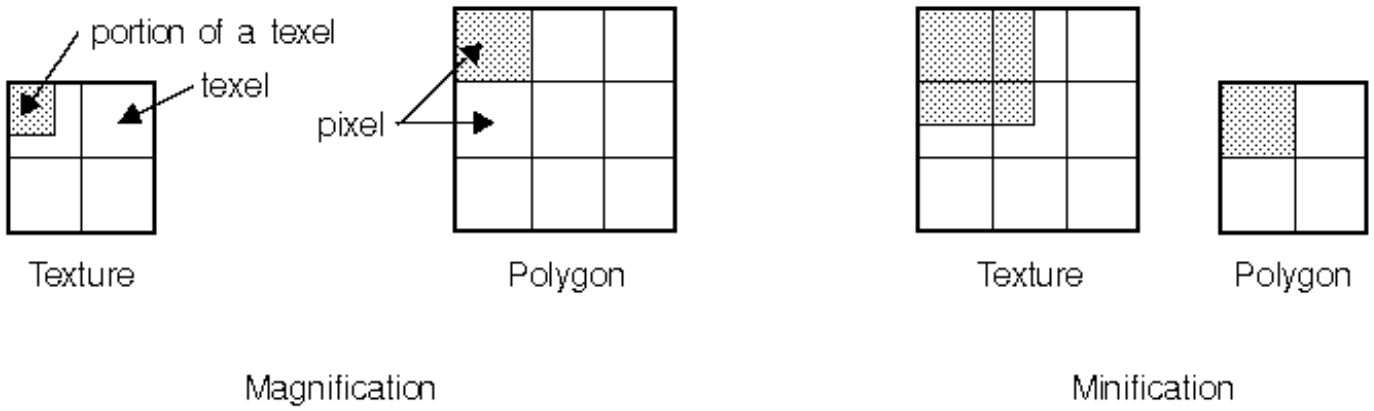
Image credit: http://www.opengl-tutorial.org/

Texture mapping pipeline

- **Compute object space location**
  - Texture coordinates are defined in the object space to move texture along with the object
- **Projector function**
  - Mapping 3D space to 2D parameter space
  - (x,y,z) → (u,v,(w)), u, v are [0 to 1]
- **Corresponder function**
  - Mapping from 2D parameter space to texture space
  - (u, v) → (i, j); 256x256 image, i, j are [0 to 255]
- **Value Transfer (Blending) function**
  - Transfer texture values into the fragment

Texture filtering

- The resolution of texture and the fragment is not 1:1 mapping
  - Magnification → 1 texel : m fragment
  - Minification → n texel : 1 fragment



[http://fly.cc.fer.hr/~unreal/theredbook/chapter09.html]

Texture filtering

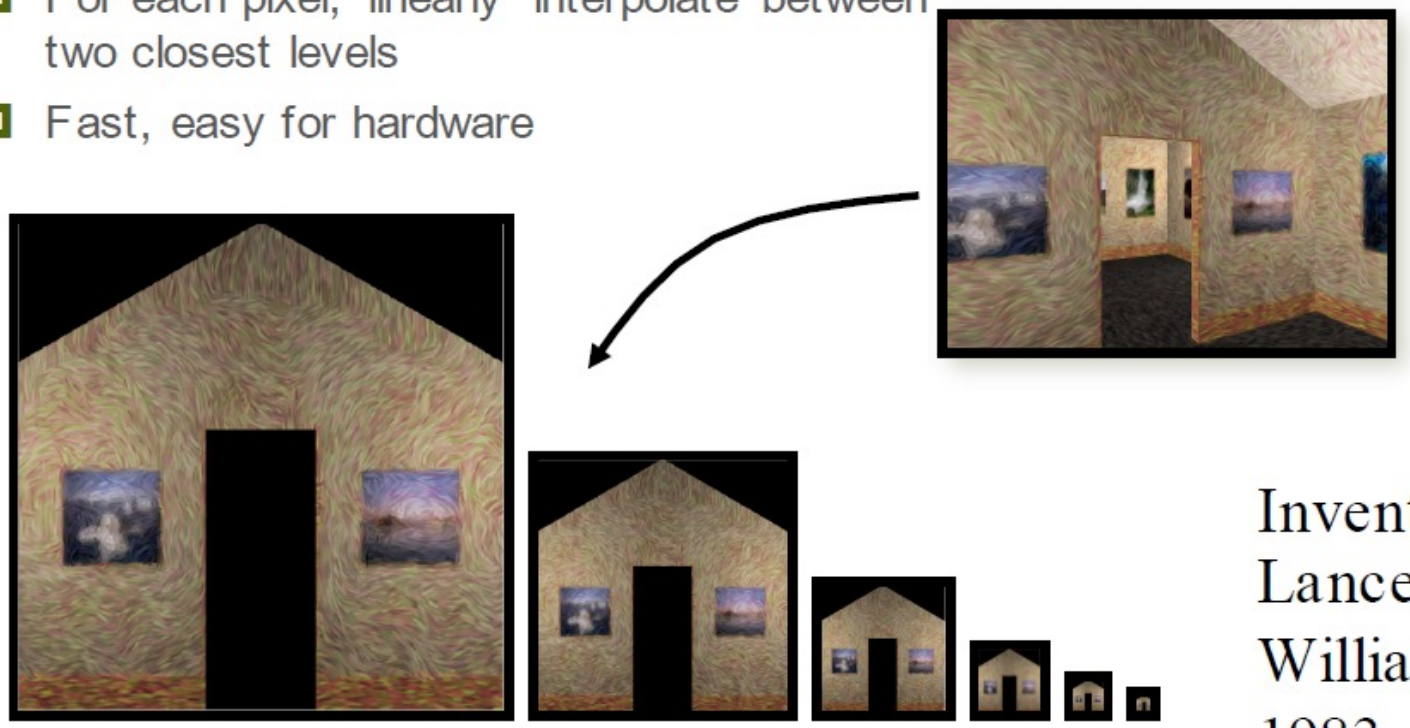
- Nearest neighbor: pick the nearest value
  - Fast but shows blocky artifacts (pixelation)
- Interpolate texel : bilinear, bicubic interpolation

Nearest neighbor | Bilinear interpolation | Bicubic interpolation

[Image by Real-time Rendering Book]

Mip Maps

- Keep textures prefiltered at multiple resolutions
  - For each pixel, linearly interpolate between two closest levels
  - Fast, easy for hardware



Invented by Lance Williams, 1983

multim im parvo (latin) - many things in a small space

Specifying a Texture in OpenGL

```
// Create one OpenGL texture
unsigned int texture;
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
// set the texture wrapping/filtering options (on the currently bound texture object)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// load and generate the texture
int width, height, nrChannels;
unsigned char *data = load_texture_data("Texture.png", &width, &height, &nrChannels, 0);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
```

glGenTextures(): takes as input how many textures we want to generate and stores them in a unsigned int array given as its second argument

glBindTexture(): Lets you create or use a named texture

glTexParameteri() sets the texture wrapping/filtering options (on the currently bound texture object

glTexImage2D() generates a texture using the previously loaded image data

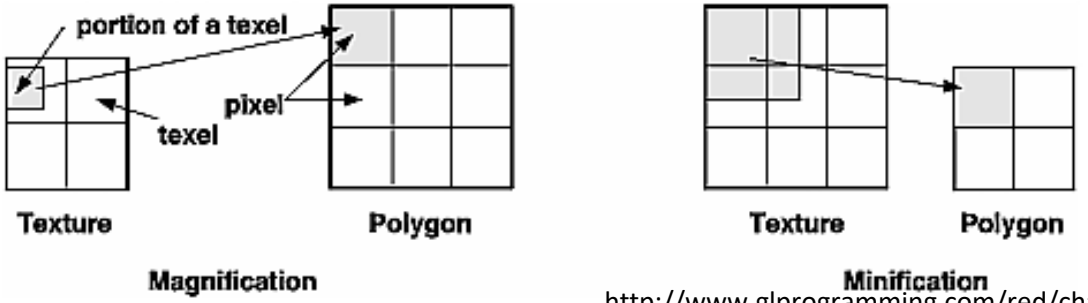
Specifying a Texture

- *void **glTexImage2D** (GLenum target, GLint level, GLint internalFormat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *texels);*
  - Specify 2D Texture
  - Target: GL_TEXTURE_2D, GL_TEXTURE_CUBE_MAP_POSITIVE_X,
  - Level: multiple level of resolution (MIPMAP)
  - Internal format: texel component (e.g. GL_RGBA)
  - Border: width of border
  - Texel: pointer storing texture image
  - Refer the details for OpenGL Programming Guide

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml

Filtering

- Texture need to be magnified or minified



http://www.glprogramming.com/red/chapter09.html

- ***glTexParameter*()***
  - Mag Filter: glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    - GL_NEAREST or GL_LINEAR
  - Min Filter: glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
    - GL_NEAREST, GL_LINEAR
    - GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR
    - GL_LINEAR_MIPMAP_NEAREST, GL_LINEAR_MIPMAP_LINEAR

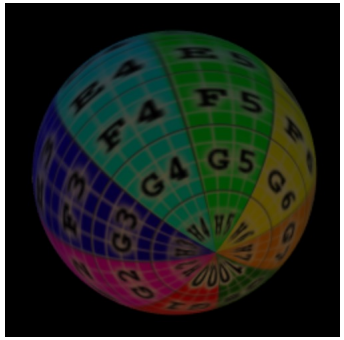
Texture Object

- Creating new texture is slow.
- Binding existing texture object is faster
  - Generate texture names
  - Initially bind texture objects to texture data
  - Bind and rebind texture objects to render object models
  - If your system cannot support many textures, establish priorities for the texture object

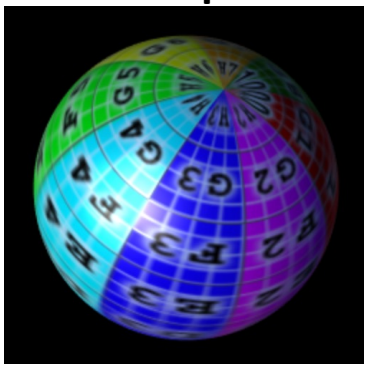
Applying textures

```glsl
#version 330 core

// Interpolated values from the vertex shaders
in vec2 UV;

// Ouput data
out vec3 color;

// Values that stay constant for the whole mesh.
uniform sampler2D myTextureSampler;

void main(){

    // Output color = color of the texture at the specified UV
    color = texture( myTextureSampler, UV ).rgb;
}
```

Diffuse



Diffuse + specular



Accepts texture UV coordinates

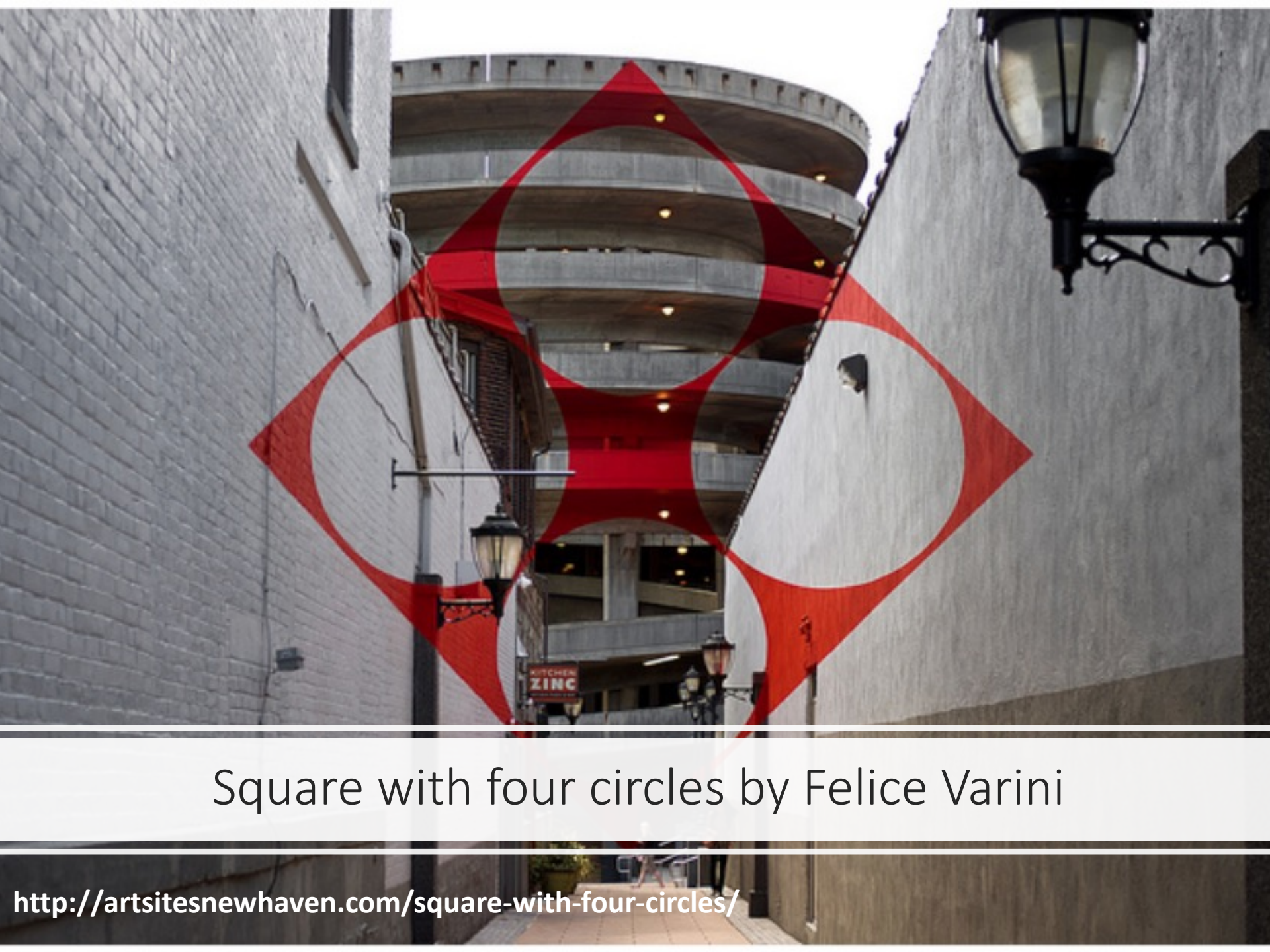
GLSL has a built-in data-type for texture objects called a sampler (in order to know which texture to access). Can have multiple samplers!

Sampling of the colour using GLSL's built-in **texture()** function using the texture parameters we set earlier

Difficulties with textures

- Tedious to specify texture coordinates for every triangle
- Textures are attached to the geometry
- Easier to model variations in reflectance than illumination
- Can't use just any image as a texture
- Projective distortions

http://www.2loop.com/3drooms.html



Square with four circles by Felice Varini

http://artsitesnewhaven.com/square-with-four-circles/

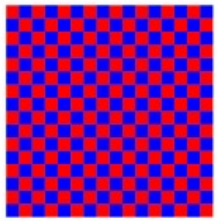
Adding Texture Mapping to Illumination

Texture mapping can be used to alter some or all of the constants in the illumination equation.

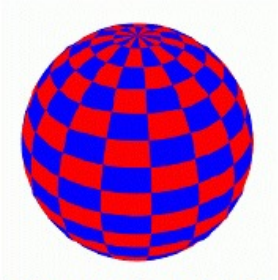
$$I = k_a L_a + k_d L_d \left(\hat{N} \cdot \hat{L}\right) + k_s L_s \left(\hat{V} \cdot \hat{R}\right)^n$$

Phong's Illumination Model

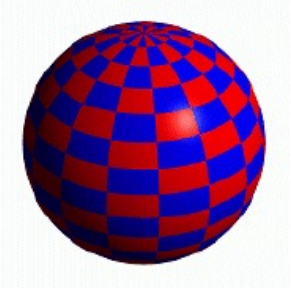
Constant Diffuse Color



Diffuse Texture Color



Texture used as Label



Texture used as Diffuse Color

Next

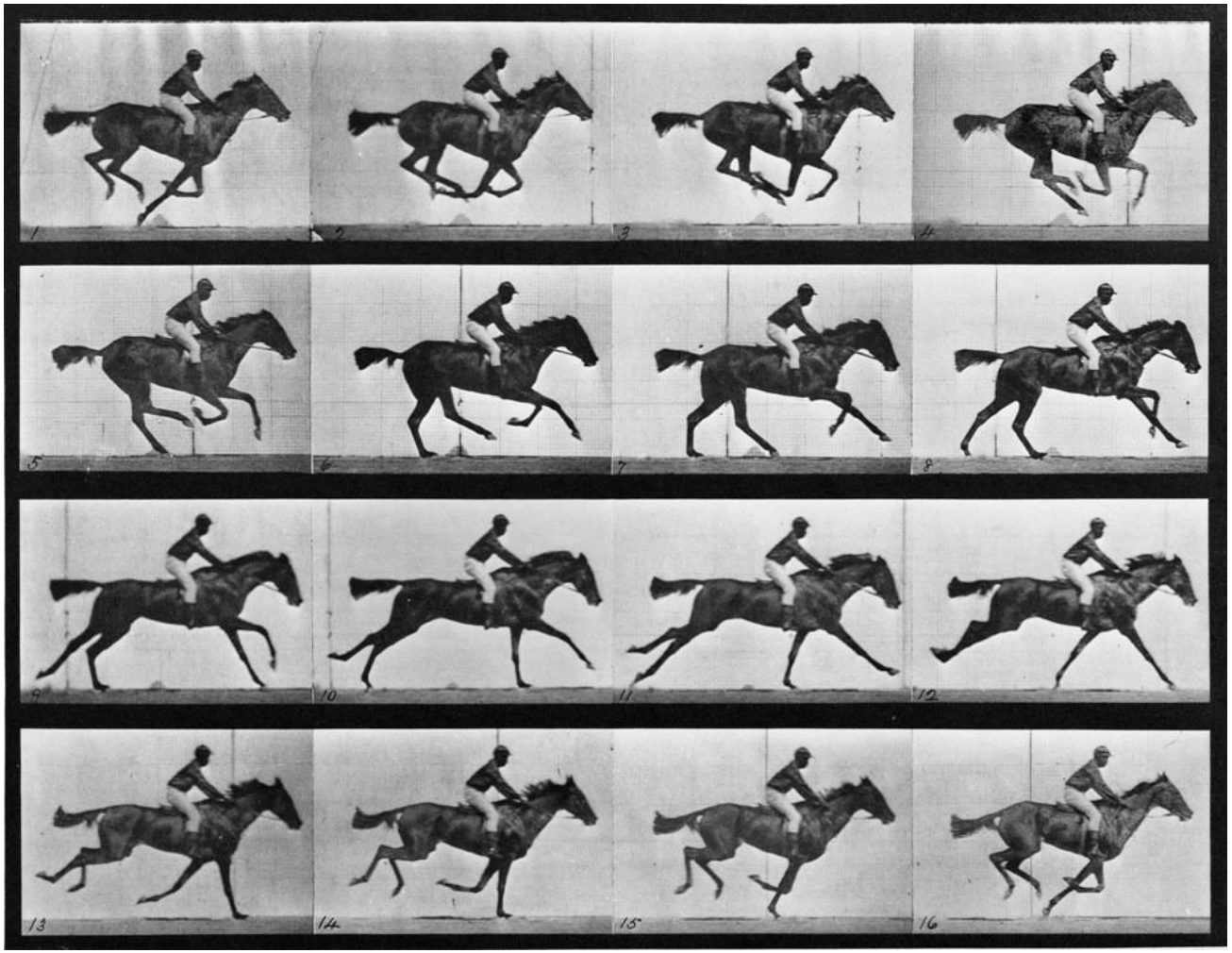
- **Intro to Animations (topic of Assignment #3)**

Perception of Motion

- How is it even possible to make it look things look like they are moving, or even alive with the computer?

Image credit: https://giphy.com/gifs/animated-loop-walking-XGnWMiVXL87Xa

Freeze frame



https://americanhistory.si.edu/muybridge