Shadow Mapping

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

- Lighting makes an 3D image looks more realistic but no shadows
- Fake shadows helps but only on a flat plane (not on objects)
- Real shadows is the best but how to create them?
- We are going to use what we have learned together with what OpenGL provides to create realistic shadows

What we have learned so far?

- Rendering to screen:
  - Draw objects on the screen

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
:
glDrawArrays(GL_TRIANGLES, ...);
glutSwapBuffer();
```

- Phong's Lighting Model
  - How to calculate the color of a fragment

```
vec4 N = ...;  // Normal of a fragment
vec4 V = ...;  // Vector from a fragment to the viewer
vec4 L = ...;  // Vector from a fragment to the light
vec4 R = ...;  // Vector (perfectly reflected ray)
// Calculate ambient, diffuse, and specular
gl_FragColor = ...;
```

  - Also need material, light color, etc

What we have learned so far?

- Use texture instead of colors
  - Need an array of texels (image)

    ```
    GLfloat my_texels[width][height][3];
    ```

  - Need a texture coordinate for each vertex (that wants to use texture)

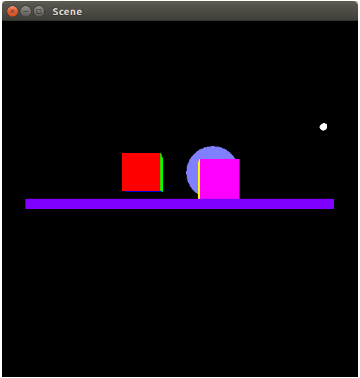
    ```
    vec2 tex_coords = {{0.0, 0.0}, {0.0, 1.0}, ...};
    ```

  - Use texture coordinate and texture in the fragment shader

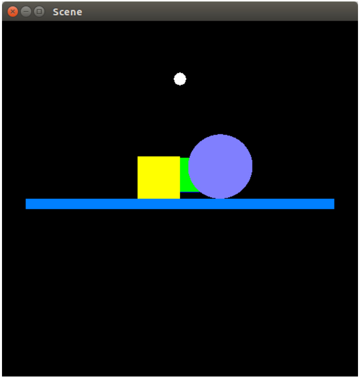
    ```
    gl_FragColor = texture2D(myTextureSampler, texCoord);
    ```

- Model view, projection, and transformation matrices

- For our discussion purpose, we are going to use this scene:



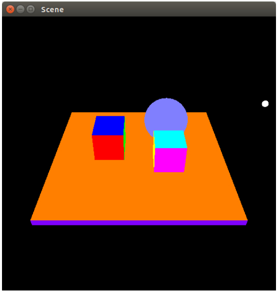
Front View



Side View

Objects in the Scene

- The floor:
  - A $3 \times 3 \times 0.1$ cube
  - The **top** part is on the plane $y = 0$
- The cube on the left
  - A $0.5 \times 0.5 \times 0.5$ cube
  - The **bottom** part is on the plane $y = 0.1$
  - The center of mass is at $(-0.5, 0.35, 0.0)$
- The cube on the right
  - A $0.5 \times 0.5 \times 0.5$ cube
  - The **bottom** part is on the plane $y = 0$
  - The center of mass is at $(0.5, 0.25, 0.25)$
- The colored sphere
  - A sphere of radius 0.4
  - Sits right on top of the floor
  - The center of mass is at $(0.5, 0.4, -0.5)$
- The Light (white sphere)
  - The center of mass is at $(2.0, 1.0, 0.0)$



Shader Programs

- Vertex Shader

```
#version 120

attribute vec4 vPosition;
attribute vec4 vColor;
attribute vec4 vNormal;
attribute vec2 vTexCoord;

varying vec4 color;
varying vec4 normal;
varying vec2 texCoord;

uniform mat4 ctm;
uniform mat4 model_view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * model_view * ctm * vPosition;
    color = vColor;
    normal = vNormal;
    texCoord = vTexCoord;
}
```

Shader Programs

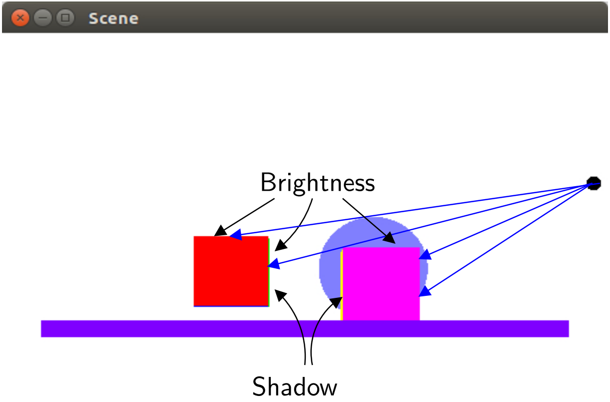
- Fragment Shader (without Phong's model)

```
#version 120

varying vec4 color;
varying vec4 normal;
varying vec2 texCoord;

uniform sampler2D myTextureSampler;
uniform int useTexture;

void main()
{
    if(useTexture == 1)
        gl_FragColor = texture2D(myTextureSampler, texCoord);
    else
        gl_FragColor = color;
}
```

- The uniform variable `useTexture` allows the application to choose how `gl_FragColor` is assigned

Shadow

- By definition, a **shadow** is a dark area where **light** from a light source is blocked by an opaque object
- So, from our scene (front view)

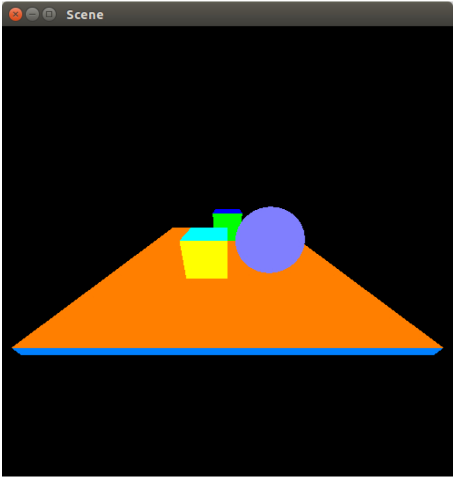


Brightness

- Suppose **Brightness** is an antonym of **Shadow**, we can define brightness as
  - **Brightness** is a bright area where **light** from a light source is not blocked by an opaque object
- So, let's imagine that a light source is a person
  - **Brightnesses** are areas that it can see
  - **Shadows** are areas that it cannot see

The Scene from the Light Point-of-View

- Here is the same scene but from the light point-of-view



- All areas that we see above should be bright
- How to create the above image?

Rendering the Scene from the Light POV

- To render the scene from the light point-of-view we need to set **model view** and **projection** matrices
- The model view matrix is quite straightforward:
  - **Eye** point is the light location
  - **At** point should be somewhere near the center of the scene
  - **Up** point is the y-axis as usual
- The projection matrix depends of the type of light source
  - For directional light (far away source), use **orthographic** projection
  - For a point source, use **perspective** projection
  - Make sure it sees the whole scene unless your light source is a spot light

Shadow by Fragment

- From the computer screen
  - The smallest unit is the pixel
  - The color of each pixel depends on fragments at that pixel
- One fragment may be in front of another fragment
  - The color of the fragment in the back is eliminated (hidden surface removal)
- From the light point-of-view
  - A fragment is bright if the light can see it (front most fragment)
  - A fragment is dark if the light cannot see it (behind other fragments)
- In fragment shader, there is a predefined variable of type `vec4` named `gl_FragCoord`
  - It keeps the coordinate of the fragment it is currently processing

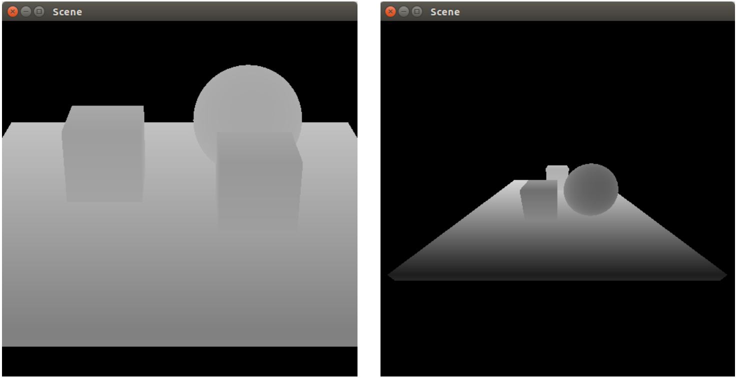
gl_FragCoord

- There is a variable named `gl_FragCoord`
  - Only available in fragment shaders
  - Automatically assign a value (per fragment) by the graphic pipeline
  - The type is `vec4` containing information about the fragment being processed:
    - x is the **screen** coordinate (0 to screen width in pixels)
    - y is the **screen** coordinate (0 to screen height in pixels)
    - z is the **depth** (between 0.0 and 1.0)

Let's see the distance

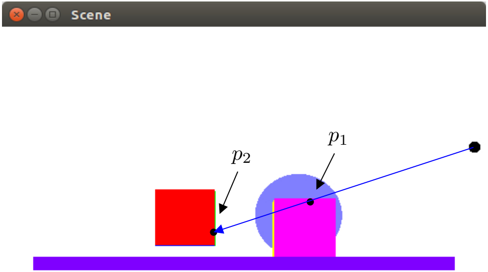
- Change the fragment shader to the following:

```
gl_FragColor = vec4(gl_FragCoord.z, gl_FragCoord.z,
                    gl_FragCoord.z, 1.0);
```



Fragment Information

- Consider the following scene:



- `gl_FragCoord.z` of p_1 is less than `gl_FragCoord.z` of p_2

Creating a Shadow for Each Fragment

- Before finalize the color of each fragment:
  - If it is blocked by other fragment, shadow
  - Otherwise, brightness
- Need to compare the distance with other fragments
  - Closest to the light, brightness
  - Otherwise, shadow
  - Unfortunately, the fragment shader only have the information about the current fragment
- Information can be sent directly to a **fragment shader**:
  - from an application using uniform variables
  - from a vertex shader using `varying` variables
  - Do not forget that a **texture** also contains information
    - from an application in a form of a texels

Textures

- Recall textures discussed in class
  - Create an empty array of texels

```
GLuint my_texels[width][height][3];
```

  - Fill `my_texels` with data
    - Using an image
    - Computer generated
  - Then transfer `my_texels` to the graphic pipeline
- We can also generate a texture inside the graphic pipeline

Render To Texture

- So far, we rendered images onto our screen
- We can also render an image onto a texture
  - Recall that a texture is a two-dimensional image
  - OpenGL we see on the screen is also a two-dimensional image
- What are we going to do are the following:
  1. Allocate memory in the graphic pipeline for a texture
  2. Render an image onto the allocated memory (not on the screen)
  3. Use the generated texture as if it is a texels

Framebuffer

- In general, when we display something on the screen, we write data to a framebuffer
- So, to render an image into a texture, we need to create a framebuffer that we can render to

```
GLuint frame_buffer_name;
glGenFrameBuffers(1, &frame_buffer_name);
glBindFrameBuffer(GL_FRAMEBUFFER, frame_buffer_name);
```

- Recall that the behavior of an OpenGL program is the same as a finite state machine
  - Anything we do after `glBindFrameBuffer()` will be for the framebuffer

Framebuffer

- We also need to allocate memory for the framebuffer

```
GLuint rendered_texture;
glGenTextures(1, &rendered_texture);

// Set the current texture
glBindTexture(GL_TEXTURE_2D, rendered_texture);

// Create an empty texture 512 x 512
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 512, 512, 0, GL_RGB,
             GL_UNSIGNED_BYTE, 0);

// Filtering
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

- Same as texture from an image except that there is no data
  - The last argument of `glTexImage2D()` is 0 instead of an array of texels
- No explicit array of texels

Depth Buffer

- Most textures are 2D image of real 3D objects/scenes
  - No hidden surface removal is required
- When we render a 3D scene, the depth buffer is used for hidden surface removal
  - A fragment in the front may be overwritten by a fragment in the back
- If we need a depth buffer, we have to create one as well

```
GLuint depth_render_buffer;
glGenRenderBuffers(1, &depth_render_buffer);
glBindRenderBuffer(GL_RENDERBUFFER, depth_render_buffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
                      512, 512);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, depth_render_buffer);
```

Depth Buffer

- Configure the new framebuffer

```
// Set rendered texture to attachment number 0
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                     renderedTexture, 0);
```

- Always check the status of the framebuffer

```
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
                              GL_FRAMEBUFFER_COMPLETE)
    return false;
```

Rendering to the Texture

- Simply bind a framebuffer and draw
  - Render to the texture:

```
glBindFramebuffer(GL_FRAMEBUFFER, frame_buffer_Name);
:
glDrawArrays(...);
```

  - Render to the screen as usual:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
:
glDrawArrays(...);
```

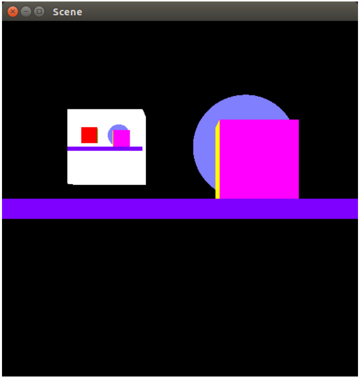
Fragment Shader

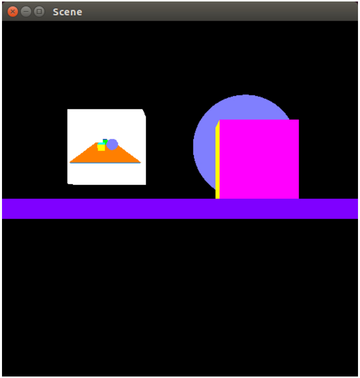
- Need to be able to choose what would be our gl_FragColor:

```
#version 120

varying vec4 color;
varying vec4 normal;
varying vec2 texCoord;

uniform sampler2D myTextureSampler;
uniform int useTexture;

void main()
{
    if(useTexture == 1)
        gl_FragColor = texture2D(myTextureSampler, texCoord);
    else
        gl_FragColor = color;
}
```

- gl_FragColor can come from either a texture or a color

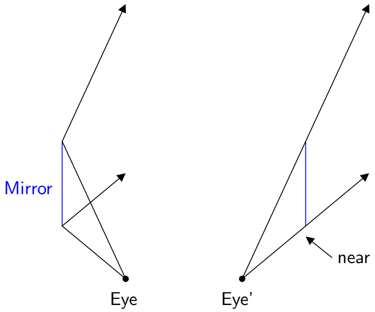
- Examples





Application

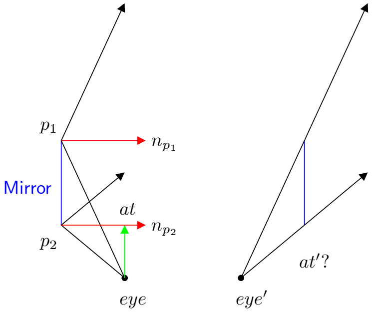
- An important application of render to texture is mirror
  - Mirror reflexes everything
  - A mirror surface should be rendered as what a viewer would see in that mirror



- It is simply a frustum with new eye location

Practice

- Consider this situation:



- Given four locations, p_1, p_2, eye, and at
- What are location of eye' and at'?
- Frustum? ($left'$, $right'$, $bottom'$, top', $near'$, and far')

Finding eye'

- From Phong's model, to calculate the color a point $\mathbf{p}$:
  - $\mathbf{l}$ is the vector from $\mathbf{p}$ to the light source
  - $\mathbf{n}$ is the plane normal at point $\mathbf{p}$
  - $\mathbf{r}$ is the direction that a perfectly reflected ray from $\mathbf{l}$ would take
- Given normalized vectors $\mathbf{l}$ and $\mathbf{n}$

$$\mathbf{r} = 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n} - \mathbf{l}$$

Finding eye'

- If you think that eye is a light source and p_2 is the point of interest
  - The vector from p_2 to eye is $eye - p_2$ and

$$\mathbf{l} = \frac{1}{|eye - p_2|}(eye - p_2)$$

- The plane normal $\mathbf{n}$ of p_2 is n_{p_2}
- From reflection, we have $\mathbf{r} = 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n} - \mathbf{l}$
  - $\mathbf{r}$ is not normalized
  - $\mathbf{r}$ only tells the **opposite** direction to eye'
  - Magnitude must be the same as the vector $eye - p_2$

Finding eye'

- Let $\mathbf{r}'$ be

$$\mathbf{r}' = -\frac{1}{|\mathbf{r}|}\mathbf{r}$$

  the normalized version of $\mathbf{r}$ in the opposite direction
- To make its magnitude the same as $eye - p_2$ simply

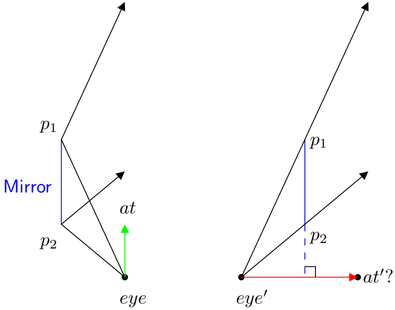
$$\mathbf{r}'' = |eye - p_2|\mathbf{r}'$$

- Thus, the point eye' can be calculated as

$$eye' = p_2 + \mathbf{r}''$$

Finding at'

- How about at'



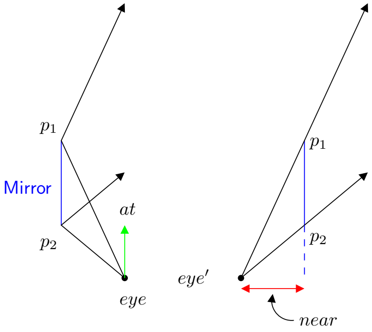
- at' can be any points where $at' - eye'$ is perpendicular to $p_1 - p_2$
- In other word,

$$(at' - eye') \cdot (p_1 - p_2) = 0$$

  - Make sure at at' is not a zero vector
  - Be careful with the direction

Finding the Frustum

- What about the frustum



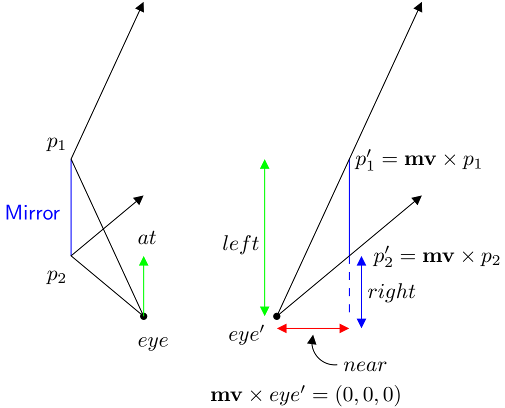
- Recall that a frustum is defined after applying the model view matrix
- In other word, a frustum is based on a camera frame
  - But p_1, p_2, eye', and at' are in the world frame

Finding the Frustum

- Earlier, eye' and at' has been calculated
- Let

$$\mathbf{mv} = \texttt{look\_at}(eye', at', up)$$

- Suppose this is a top view:



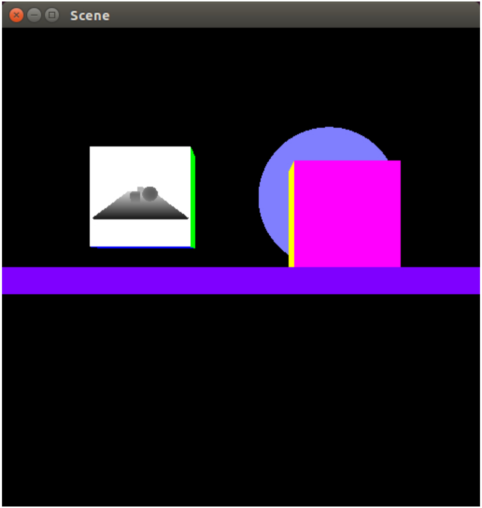
Render gl_FragCoord.z

- We can also render gl_FragCoord.z as a texture
- Consider the following fragment shader:

```
#version 120

varying vec4 color;
varying vec4 normal;
varying vec2 texCoord;

uniform sampler2D myTextureSampler;
uniform int useTexture;
uniform int render_to_texture;

void main()
{
    if(render_to_texture == 1)
        gl_FragColor = vec4(gl_FragCoord.z, gl_FragCoord.z,
                            gl_FragCoord.z, 1.0);
    else if(useTexture == 1)
        gl_FragColor = texture2D(myTextureSampler, texCoord);
    else
        gl_FragColor = color;
}
```

Render gl_FragCoord.z

- Here is a result on the surface of a cube from the light point-of-view



Information in the Texture

- Each fragment is a color

```
gl_FragColor = vec4(gl_FragCoord.z, gl_FragCoord.z,
                    gl_FragCoord.z, 1.0);
```

- Given a texture coordinate `texCoord`:

```
texture2D(myTextureSampler, texCoord).z
```

  gives us the distance of the closest fragment at `texCoord` to the light position from the light point-of-view
- When we are rendering onto the screen, we know the depth of each fragment from `gl_FragCoord`
  - Unfortunately, it is the depth from the eye point-of-view (not the light position)

Finding a Vertex's Texture Coordinate

- Recall the scene from the light source point-of-view:
  - Model view and projection matrix are required
- Let
  - `vPosition` be a vertex position, and
  - `light_model_view` and `light_projection` be the model view and projection matrices of a light source

  The new location of `vPosition` (according to the light POV)

```
vec4 newPosition = light_projection * light_model_view * vPosition;
```

- `newPosition` is the location of `vPosition` according to the light POV
  - Only in a vertex shader
  - `newPosition.w` is not 1.0 (perspective projection)

Finding the Texture Coordinate

- Let `fragPosition` in a fragment shader be an interpolated `newPosition` from a vertex shader and

```
vec4 newFragPosition = fragPosition / fragPosition.w;
```

  - `newFragPosition.w` is now 1.0.
- Object from the light point-of-view is visible if
  - $-1.0 \leq$ `newFragPosition.x` ≤ 1.0,
  - $-1.0 \leq$ `newFragPosition.y` ≤ 1.0, and
  - $-1.0 \leq$ `newFragPosition.z` ≤ 1.0.
- Recall that a texture coordinate (s, t):
  - $0.0 \leq s \leq 1.0$
  - $0.0 \leq t \leq 1.0$
- Range of `newFragPosition.x` and `newFragPosition.y` must be modified

- Mapping to a new range
  - $0.0 \leq s \leq 1.0$ but $-1.0 \leq \texttt{newFragPosition.x} \leq 1.0$

$$
\begin{aligned}
-1.0 \leq& \texttt{newFragPosition.x} \leq 1.0 \\
-1.0 + 1.0 \leq& \texttt{newFragPosition.x} + 1.0 \leq 1.0 + 1.0 \\
0.0 \leq& \texttt{newFragPosition.x} + 1.0 \leq 2.0 \\
0.0/2.0 \leq& (\texttt{newFragPosition.x} + 1.0)/2.0 \leq 2.0/2.0 \\
0.0 \leq& (0.5 \times \texttt{newFragPosition.x}) + (0.5 \times 1.0) \leq 1.0
\end{aligned}
$$

    - Let $s = (0.5 \times \texttt{newFragPosition.x}) + (0.5 \times 1.0)$
  - Similarly, $0.0 \leq t \leq 1.0$ but $-1.0 \leq \texttt{newFragPosition.y} \leq 1.0$
    - $t = (0.5 \times \texttt{newFragPosition.y}) + (0.5 \times 1.0)$

- Recall that $0.0 \leq \texttt{gl\_FragCoord.z} \leq 1.0$ where
  - 0.0 is the closest
  - 1.0 is the farthest
- But $-1.0 \leq \texttt{newFragPosition.z} \leq 1.0$ where
  - 1.0 is the closest and
  - -1.0 is the farthest
- If we use the same calculation:

$$u = (0.5 \times \texttt{newFragPosition.z}) + (0.5 \times 1.0)$$

  - If u = 0.0, it is the farthest
  - If u = 1.0, it is the closest
- Thus, we need

$$\begin{aligned} u =& 1 - ((0.5 \times \texttt{newFragPosition.z}) + (0.5 \times 1.0)) \\ & (-0.5 \times \texttt{newFragPosition.z}) + (1 - 0.5) \\ & (-0.5 \times \texttt{newFragPosition.z}) + (0.5 \times 1.0) \end{aligned}$$

Finding the Texture Coordinate

- So, we have

$$\begin{bmatrix} s \\ t \\ u \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.0 & 0.0 & 0.5 \\ 0.0 & 0.5 & 0.0 & 0.5 \\ 0.0 & 0.0 & -0.5 & 0.5 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1.0 \end{bmatrix}$$

  where
  - (s, t) is a texture coordinate of the transformed vertex (x, y, z)
  - u is the distance of the transform vertex to the viewer where
- For better performance, the matrix

$$\begin{bmatrix} 0.5 & 0.0 & 0.0 & 0.5 \\ 0.0 & 0.5 & 0.0 & 0.5 \\ 0.0 & 0.0 & -0.5 & 0.5 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \times \texttt{light\_projection} \times \texttt{light\_model\_view}$$

  should be sent as a uniform variable to the vertex shader

Shader Programs

- In the vertex shader:

```
uniform mat4 light_model_view;
uniform mat4 light_projection;
varying vec4 newPosition;
:
void main()
:
    newPosition = light_projection * light_model_view * vPosition;
:
```

- In the fragment shader:

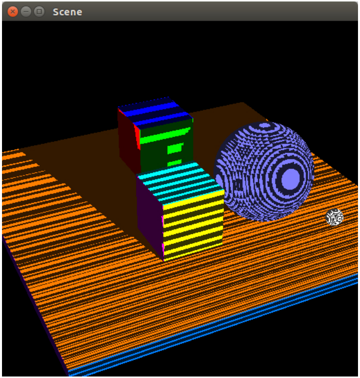
```
varying newPosition;
mat4 bias = mat4(vec4(0.5, 0.0, 0.0, 0.0),
                 vec4(0.0, 0.5, 0.0, 0.0),
                 vec4(0.0, 0.0, -0.5, 0.0),
                 vec4(0.5, 0.5, 0.5, 1.0));
:
void main()
:
    vec4 newFragPosition = bias * (newPosition / newPosition.w);
:
```

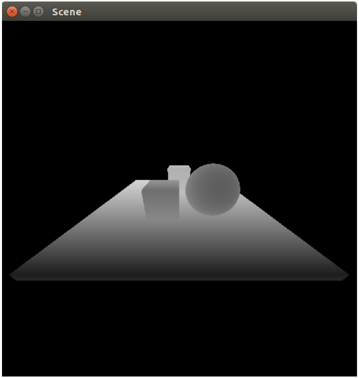
Shader Programs

- When rendering, use texture as a depth reference:

```
if(render_to_texture == 1)
    gl_FragColor = vec4(gl_FragCoord.z, gl_FragCoord.z,
                        gl_FragCoord.z, 1.0);
else
{
    vec4 newFragPosition = bias * (shadowCoord / shadowCoord.w);
    if(newFragPosition.z <=
       texture2D(myTextureSampler,
                 vec2(newFragPosition.x, newFragPosition.y)).z)
        gl_FragColor = color;
    else
        gl_FragColor = color * 0.2;
}
```

Result

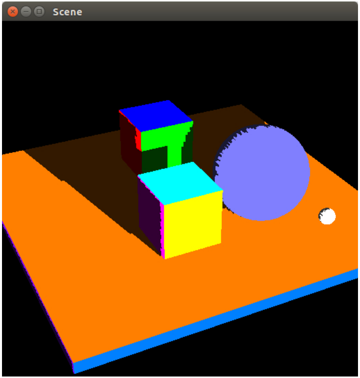




Shadow Acne

Result

- Need to add a little bit of error margin




```
if(newFragPosition.z <=
   texture2D(myTextureSampler,
             vec2(newFragPosition.x, newFragPosition.y)).z + 0.01);
:
```

More Precision

- Recall that we created RGB texture

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 512, 512, 0, GL_RGB,
             GL_UNSIGNED_BYTE, 0);
```

- Each color element is a `GLubyte` (8-bit value)
- We loose precision when we assign `gl_FragCoord.z` to a byte value
- We only need depth

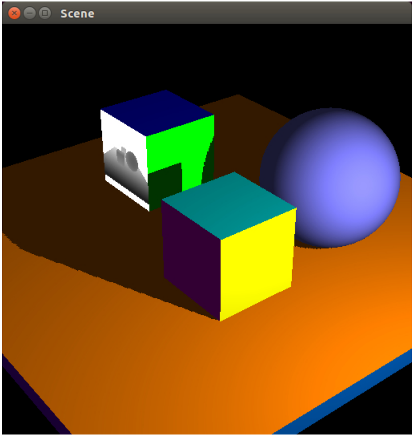
```
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT16, 512, 512,
             0, GL_DEPTH_COMPONENT, GL_FLOAT, 0);
```

  when render in the fragment shader

```
gl_FragDepth = gl_FragCoord.z;
```

Result

- Need to add a little bit of error margin




```
if(newFragPosition.z <=
   texture2D(myTextureSampler,
             vec2(newFragPosition.x, newFragPosition.y)).z + margin);
```

- The value of margin should be based on vectors L and N