CS 432 – Interactive Computer Graphics

Lecture 08 – Part 2

Shadows

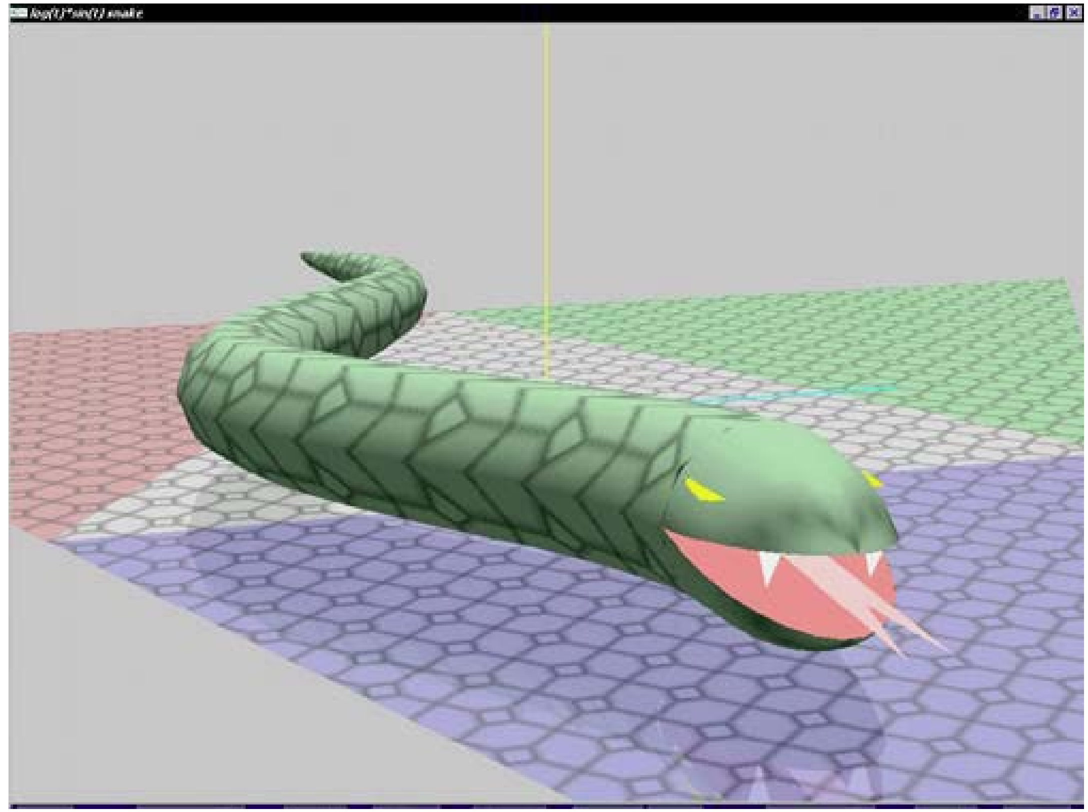
Reading

- Angel
  - Section 4.10 (pgs 249-253) – Shadows
  - Section 6.12 (pgs 342-344) – Antialiasing
- Red Book
  - Shadow Mapping – pgs 400-409

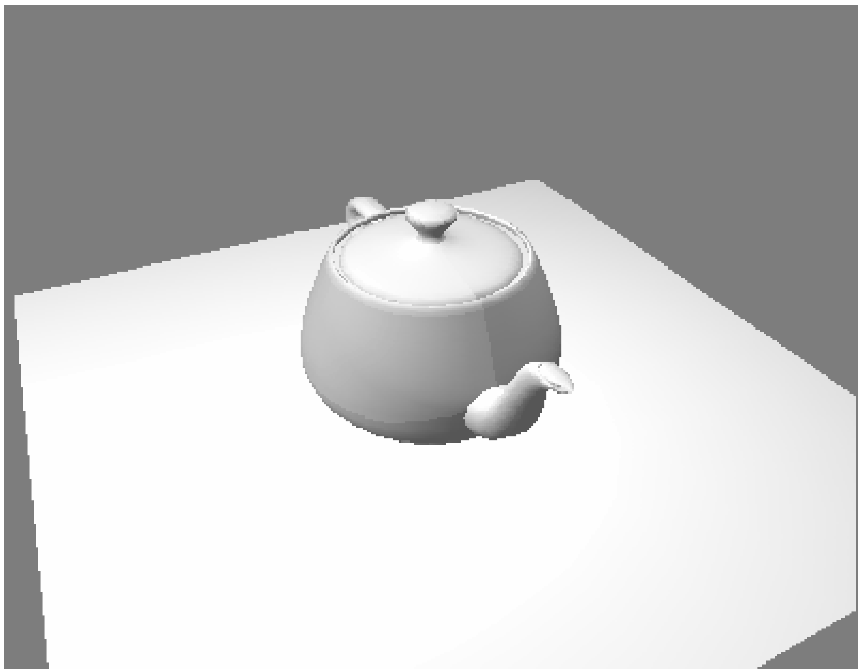
Shadows

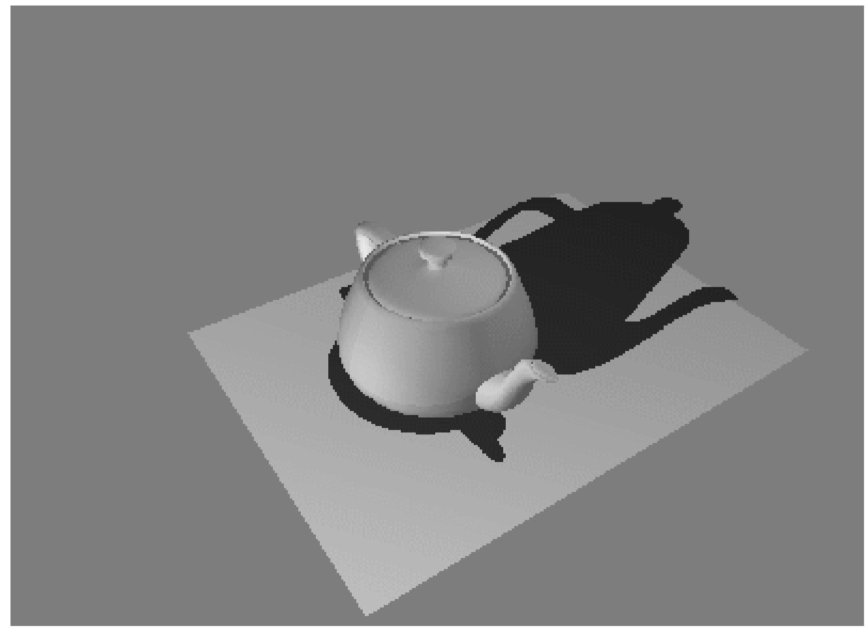
- Why do we need them?
  - Provide visual cues about the spatial relationships between different components in a scene
  - Anchors: Without shadows objects seem to hover/float
  - Improves “realism”
  - Lighting environment cues

Shadows



Hovering Objects



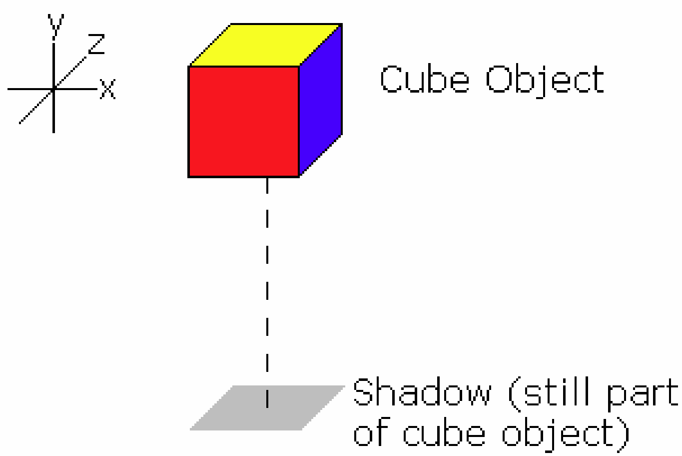


Shadow Algorithms

- Fake shadow
- Planar Projection
- Shadow Mapping (Z-buffer Algorithm)
- Lots more…

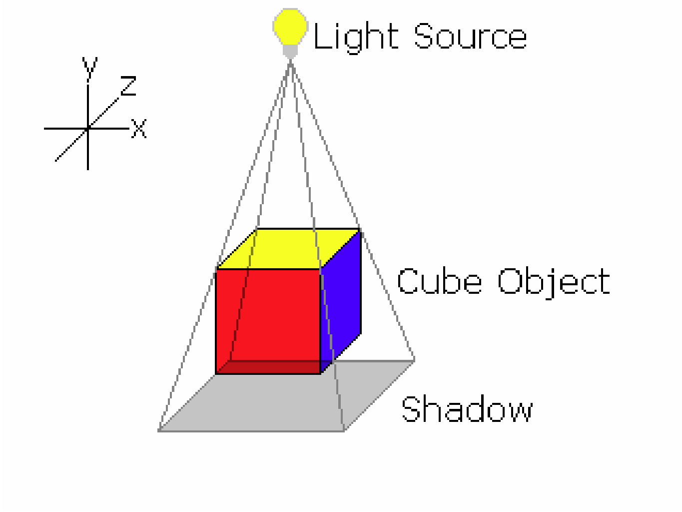
Fake Shadows

- Works only in special cases
- Basic Example
  - Project the object onto a plane (the floor)
  - Keep the shadow as part of the object description



Projection

- Project onto **each** ground plane with COP the light source
- Limited to planar receivers
- Complicated if trying to do directed lights (best with point sources)



Shadow Mapping/Z-Buffer Algorithm

- Render the scene from view point of light source
- Store the z-buffer (depth) into a “shadow map”
- Render the scene from the camera view
  - Let the T be the transform re-aligning the camera coordinate system to the coordinate system of the light source
- For each pixel in the camera view let $P = (x, y, z)$ be the 3D coordinates of the corresponding visible scene point
  - Transform the camera coordinates P into light source coordinates $Q = (u, v, w)$ where $Q = T * P$
- Let d be the “z” value of the (u, v) pixel in the shadow map
  - If $(d < w)$ then P is in the shadow now lit by this source (there is something else closer to the camera than this object, shadowing it)
  - Else P is lit by this source
- Do this for each source if multiple sources are present

Z-Buffer Algorithms

- Pro
  - Works for any object
  - API & Hardware implementations
  - Good/easy (relatively) for directional lights
- Cons
  - Could be slow
  - Aliasing problems
  - Slightly tougher for point lights

Multi-Pass Rendering

- Shadow Projection and Shadow Mapping (Z-buffer) algorithms are implemented in OpenGL by doing *multi-pass renderings*
  - Render the scene more than once
- We have already done this once
  - Environment mapping

Shadow Projection

- Let us assume that shadows fall on the ground or the surface, $y = 0$
- The shadow of an object, called the **shadow polygon**, is the projection of the original polygon onto a surface
    - Projected from the light source
      (i.e the light source is the center-of-projection,COP)

Shadow Projection

- Given a **point** light source at (x_L, y_L, z_L) the basic algorithm is:
  1. Render once as normal
  2. Translate the world (i.e each object) so that the light is at its origin
  3. Pre-multiply this by the **shadow projection** matrix that is a simple projection onto the ground plane, we'll call this $\boldsymbol{M}$

$$\boldsymbol{M} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \dfrac{\mathbf{1}}{-\boldsymbol{y_L}} & \mathbf{0} & \mathbf{0} \end{bmatrix}$$

  4. Finally we translate everything back from the light position
  5. Now render again (with our new model_matrix M) and using a simple shader that will render a "dark" color

- Note: You can easily move these resulting shadows elsewhere (not on the plane $y = 0$) by applying another transformation (like a translation to move it up or down)

Example: Shadow Projection

New Model Matrix

Error in code in Angel 6th edition

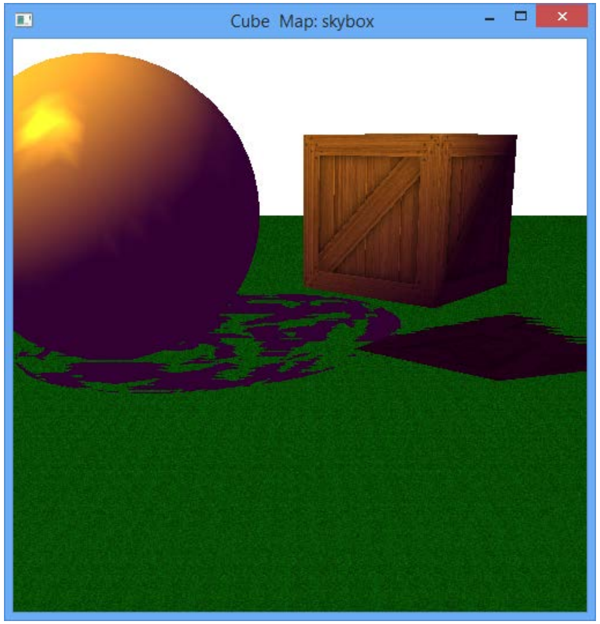
Shadow matrix

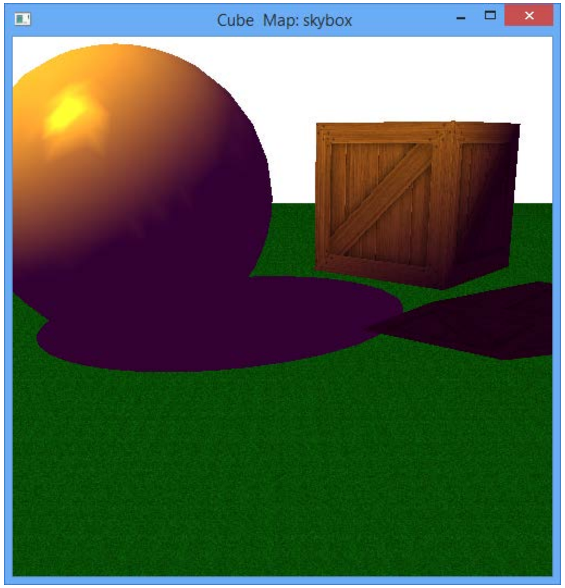
Simple Shader to draw black

```
void CubeTextured::drawProjectionShadow(Camera cam, Light light) {

    ///////////////////SHADOW STUFF//////////////////
    if (light.getPosition().y > 0) {
        mat4 shadow_Matrix = mat4(1.0);

        shadow_Matrix[3][1] = -1.0f / light.getPosition().y;
        shadow_Matrix[3][3] = 0;

        float offset = 0.01f;

        //turn light to all back color
        Light shadowLight(light.getPosition(), vec4(0, 0, 0, 1), vec4(0, 0, 0, 1), vec4(0, 0, 0, 1));

        mat4 M = Translate(light.getPosition().x, light.getPosition().y + offset, light.getPosition().z)*shadow_Matrix
            *Translate(-light.getPosition().x, -light.getPosition().y, -light.getPosition().z)*modelmatrix;

        glBindVertexArray(shadowVAO);
        glUseProgram(shadowProgram);

        GLuint MVP_loc = glGetUniformLocation(shadowProgram, "MVP_matrix");
        glUniformMatrix4fv(MVP_loc, 1, GL_TRUE, cam.getProjectionMatrix()*cam.getViewMatrix()*M);

        glDrawArrays(GL_TRIANGLES, 0, numVertices);
    }
}
```

Shadow Projection Issues

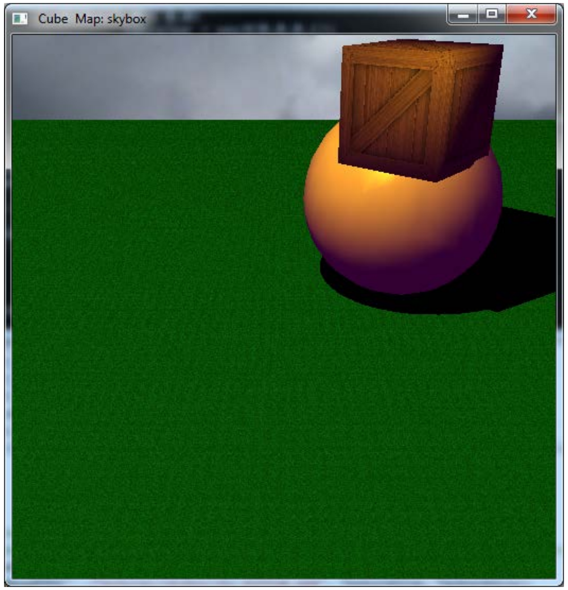
- Both the ground and the shadow are at the same depth (aliasing/precision issues cause bad rendering or “depth fighting”)
- Solutions:
  1. Blending (see later)
  2. Move shadows slightly above the ground plane (translate the y amount back by and additional ~0.01)





Shadow Projection Issues

- As mentioned there's also the issue of shadows only appearing on the planes they are projecting onto
- Also tougher for directional lights



Shadow Mapping

- Shadow mapping uses a *depth texture* to determine whether a fragment is lit or not
- It is a *multi-pass* technique
    - View the scene from the shadow-casting light source
        - Everything seen from this location should be lit
    - By rendering the scene's depth from this point-of-view into a depth buffer we can obtain a *map* of the shadowed and un-shadowed points
    - Those points visible will be rendered, those hidden will be culled away by the depth test

Shadow Mapping: Two Passes

- Pass 1
    1. Render the scene from the POV of the light source in the direction of the light
        - For point source we will need to look in all 6 directions!
    2. Create a shadow map by attaching a depth texture to a frame buffer object and rendering depth into it.
- Pass 2
    1. Render the scene from POV of the viewer.
    2. Transform coordinates into the light's reference frame and compare their depths to the depths recorded in the light's depth texture
    3. Fragments that are further from the light than the recorded depth value were not visible to the light, and hence in shadow

OpenGL Shadow Mapping

- Let's create a texture when we initialize our system
  - Actually we'll render each light to a different texture
- We also don't want to render to the screen but instead just render the depth information to some other framebuffer.

```
//stuff for shadow mapping
glGenFramebuffers(1, &depthbuffer);
for (unsigned int i = 0; i < lights.size(); i++) {
	GLuint st;
	glGenTextures(1, &st);

	//depth texture
	glBindTexture(GL_TEXTURE_2D, st);
	glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, shadowMapSize, shadowMapSize, 0,
		GL_DEPTH_COMPONENT, GL_FLOAT, 0);

	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
	glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

	shadowtextures.push_back(st);
}
```

OpenGL Shadow Mapping

- Every time we need to draw we must first create the shadow maps
- We'll do this by rendering from the light's POV in the direction of the light
  - And just rendering the depth information to the framebuffer/texture

Culling

- If we don't *cull* faces, the front face will shadow the back etc..
    - `glEnable(GL_CULL_FACE);`
- So before making shadows let's do culling on the front, then make our shadow map, then revert to culling of the back faces:

```
void display( void )
{
    glCullFace(GL_FRONT);  //only want 1 face to make shadows, and should be back face relative to the light
    generateShadowMaps();

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    skybox->draw(skyboxCamera, lights, shadowtextures);

    glCullFace(GL_BACK);
    for (unsigned int i = 0; i < objects.size(); i++) {
        if (objects[i] != skybox)
            objects[i]->draw(camera, lights,shadowtextures);
    }
    glutSwapBuffers();
}
```

Generating Shadow Maps

- So when it's time to make the shadow maps
  - Set the framebuffer to be the one we created.
  - Set the viewport to be the framebuffer's size
  - Disable drawing and reading since we're only doing depth.
- Now for each light source that we want to cast shadows:
  - Bind the desired texture to render to.
  - Bind the depth information of the current framebuffer to this texture
  - Clear out any old depth information.
  - Positional a "virtual camera" at the light with the desired projection
  - Render all the objects using this virtual camera
    - These will be rendered to the current framebuffer who's depth information will be put in the selected texture.
- Afterwards we'll want to restore to the screen's framebuffer (0) and the previous viewport.

OpenGL Shadow Mapping

Bind the framebuffer we need to render to

Should say no buffer is used for color stuff

Attach the texture to the framebuffer

Get ready to render from camera's POV

Draw!

```
void generateShadowMaps() {

    glBindFramebuffer(GL_FRAMEBUFFER, depthbuffer);
    glViewport(0, 0, shadowMapSize, shadowMapSize);
    glDrawBuffer(GL_NONE);  //which color buffers to draw to
    glReadBuffer(GL_NONE);  //select color buffer source for pixels

    Camera tempCam;
    for (unsigned int i = 0; i < lights.size(); i++) {
        glBindTexture(GL_TEXTURE_2D, shadowtextures[i]);
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, shadowtextures[i], 0);
        glClear(GL_DEPTH_BUFFER_BIT);

        if(lights[i].getType()==0)//directional
            tempCam.positionCamera(lights[i].getPosition(), normalize(-lights[i].getDirection()), vec4(0, 
        else if (lights[i].getType()==2) //spotlight
            tempCam.positionCamera(lights[i].getPosition(), normalize(-lights[i].getDirection()), vec4(0, 
        else //point
            tempCam.positionCamera(lights[i].getPosition(), normalize(lights[i].getPosition() - vec4(0, 0,
        tempCam.setProjection(Frustum(-1, 1, -1, 1, 1, 200));
        for (unsigned int j = 0; j < objects.size(); j++) {
            if (objects[j] != skybox)
                objects[j]->drawShadowMap(tempCam);
        }
    }
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glViewport(0, 0, ww, wh);
}
```

OpenGL Shadow Mapping

- What's happing in the `drawShadowMap` function?
  - Not much!
- We'll just render using a simple shader and the light's POV camera

```
void Drawable::drawShadowMap(Camera cam) {

	////////////////////SHADOW STUFF//////////////////
	glUseProgram(shadowProgram);
	//glBindVertexArray(0);
	glBindBuffer(GL_ARRAY_BUFFER, VBO);

	GLuint vPosition = glGetAttribLocation(shadowProgram, "vPosition");
	glEnableVertexAttribArray(vPosition);
	glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

	GLuint MVP_loc = glGetUniformLocation(shadowProgram, "MVP_matrix");
	glUniformMatrix4fv(MVP_loc, 1, GL_TRUE, cam.getProjectionMatrix()*cam.getViewMatrix()*modelmatrix);
	glDrawArrays(GL_TRIANGLES, 0, numVertices);

}
```

Shadow Shaders

- Very basic shaders to just compute the positions using the matrices that we specified for shading
- Vertex Shader

```
#version 150

in  vec4 vPosition;
out float pos;

uniform mat4 MVP_matrix;

void main()
{
      gl_Position = MVP_matrix*vPosition;
}
```

- Fragment Shader

```
void main()
{

}
```

No need for a color at all!
Leave it as default black

OpenGL Shadow Mapping

- Ok so now we have our shadow maps rendered into textures
  - How do we use them?
- When we render the objects as normal we need to compare each *fragment*'s depth to that in the depth maps
  - But this needs to be relative to the light
- So we'll also need a transformation matrix to take us from world coordinates into the light's *projective space* so we can check depth values
- For directional lights this would be something like:
  - `mat4 light_camera_matrix = LookAt(lights[i].getPosition(), lights[i].getPosition() + lights[i].getDirection(), vec4(0, 1, 0, 0));`
  - `mat4 light_proj_matrix = Frustum(-1, 1, -1, 1, 1, 200);`
  - `mat4 MS_Matrix = light_proj_matrix*light_camera_matrix*modelmatrix;`

OpenGL Shadow Mapping

- Vertex Shader
  - As per usual do anything you want to do with shading and texture mapping
  - But we also want to find out where the current vertex is in each shadow map
    - Just do this by multiplying the vertex by the MS_matrix we passed it and pass it to the fragment shader

```
VOs1.shadow_coord = lights1.MS_matrix*vPosition;
```

OpenGL Shadow Mapping

- Fragment Shader
  - Now that we have the shadow map texture coordinates for the fragment we need to do a few things…

1. Do perspective division on vertex to ensure that the w coordinate is one.
   - `vec3 shadowMapTexCoord = shadow_coord/shadow_coord.w`
   - The vertices that are visible should be in the range $-1 <= xyz <= 1$
2. Move vertices into texture coordinates $0 < uvw < 1$
   - `shadowMapTexCoord = 0.5*shadowMapTextureCoord + 0.5`
3. If the xyz values of this are within texture coordinates (check to make sure each of these are within [0,1]), get the depth from the depth texture
   - `float f = texture2D(depth_texture, shadow_coord.xy).z`
4. Use this to make decisions on how to color fragment
   - Based on comparing `f` and `shadowMapTexCoord.z`

Ex: Shadow Mapping – Vertex Shader

```
in vec4 vPosition;
in vec3 vNormal;
in vec2 vTexCoord;

out vec4 color;
out vec2 texCoord;
out vec4 shadow_coord;

uniform mat4 model_matrix;
uniform mat4 camera_matrix;
uniform mat4 proj_matrix;
uniform mat4 shadow_matrix;


uniform vec4 lightPos;
uniform vec4 ambientProduct, diffuseProduct, specularProduct;
uniform float alpha;


void main()
{
        texCoord = vTexCoord;
```

```
        color = (ambient+diffuse+specular)*(1/pow(dist,2));
        color.a = 1.0;


        gl_Position = proj_matrix*camera_matrix*model_matrix*vPosition;

        shadow_coord = shadow_matrix*model_matrix*vPosition;


}
```

Ex: Shadow Mapping – Fragment Shader

Since we're passing in a homogenous coordinate that underwent projection, we must do perspective division

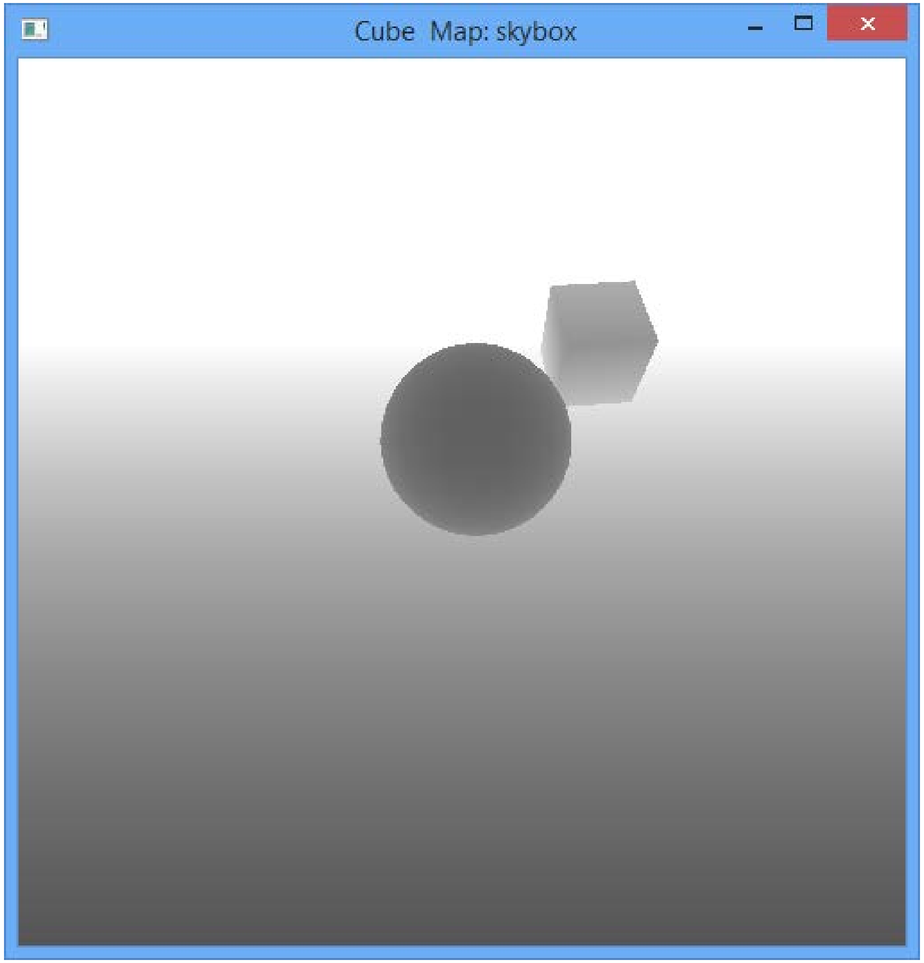
```
//whatever coloring stuff… resulting in color variable
vec3 shadowMapTexCoord = shadow_coord.xyz/shadow_coord.w;
shadowMapTexCoord = 0.5*shadowMapTexCoord+0.5;

//distance between light an nearest occluder
float distFromLight = texture(depth_texture, shadowMapTexCoord.st).r;

if(shadowMapTexCoord.z >=0 && shadowMapTexCoord.z <=1 && shadowMapTexCoord.x >= 0 &&
     shadowMapTexCoord.x <= 1 && shadowMapTexCoord.y >= 0 && shadowMapTexCoord.y <= 1){
     if(distFromLight < shadowMapTexCoord.z-0.01)
          color = vec4(0.2,0.2,0.2,1)*color;  //make it darker
     }
```

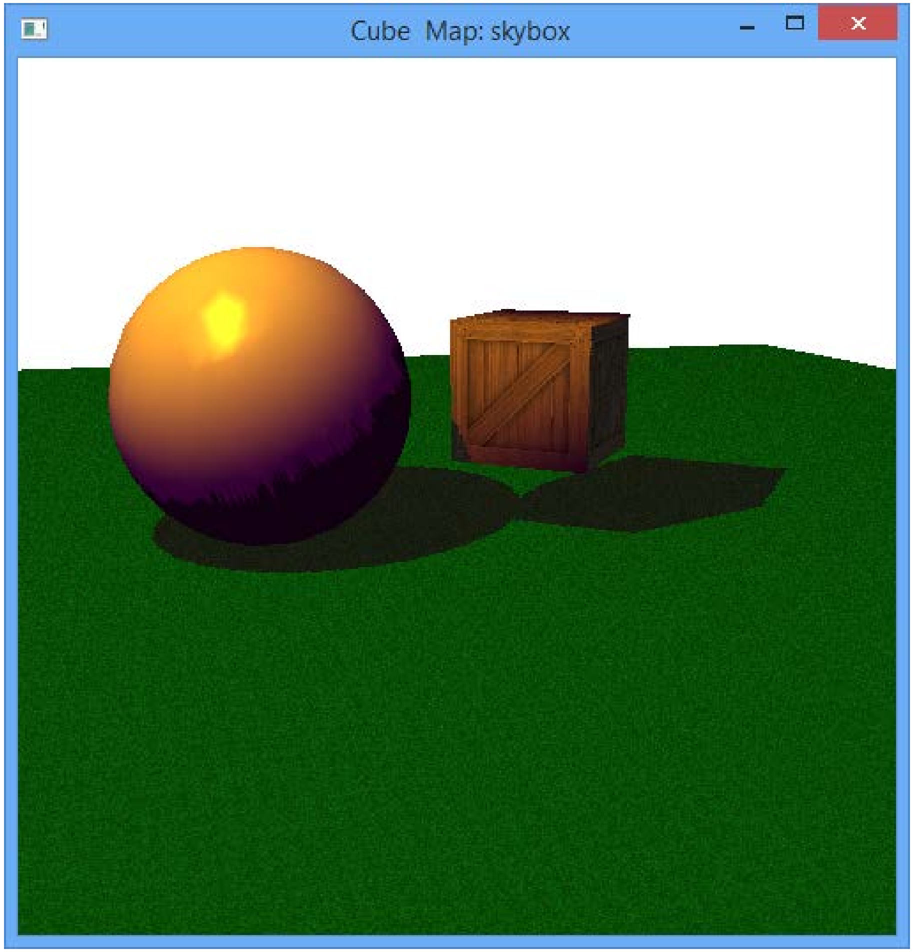
Bias to help with sampling errors

See if the nearest thing is this thing

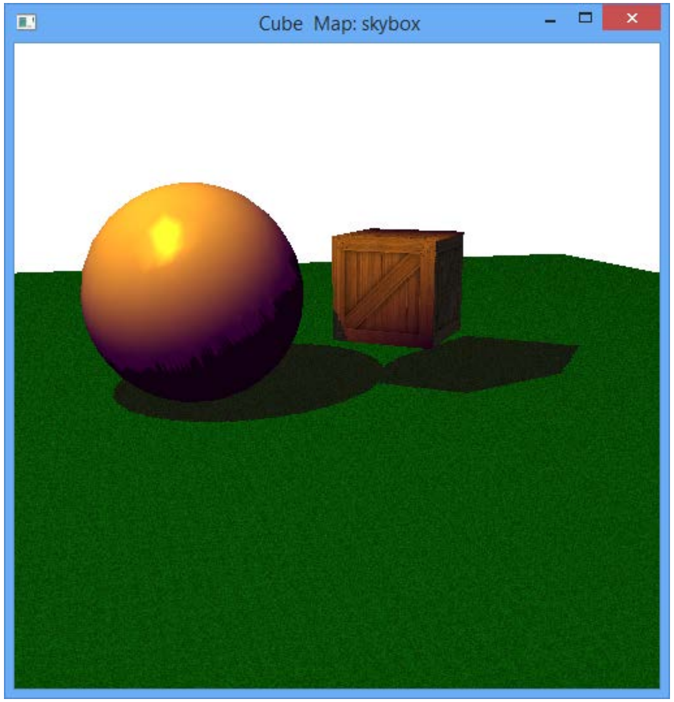
Check if point is in light's frustrum

Ex: Shadow Mapping



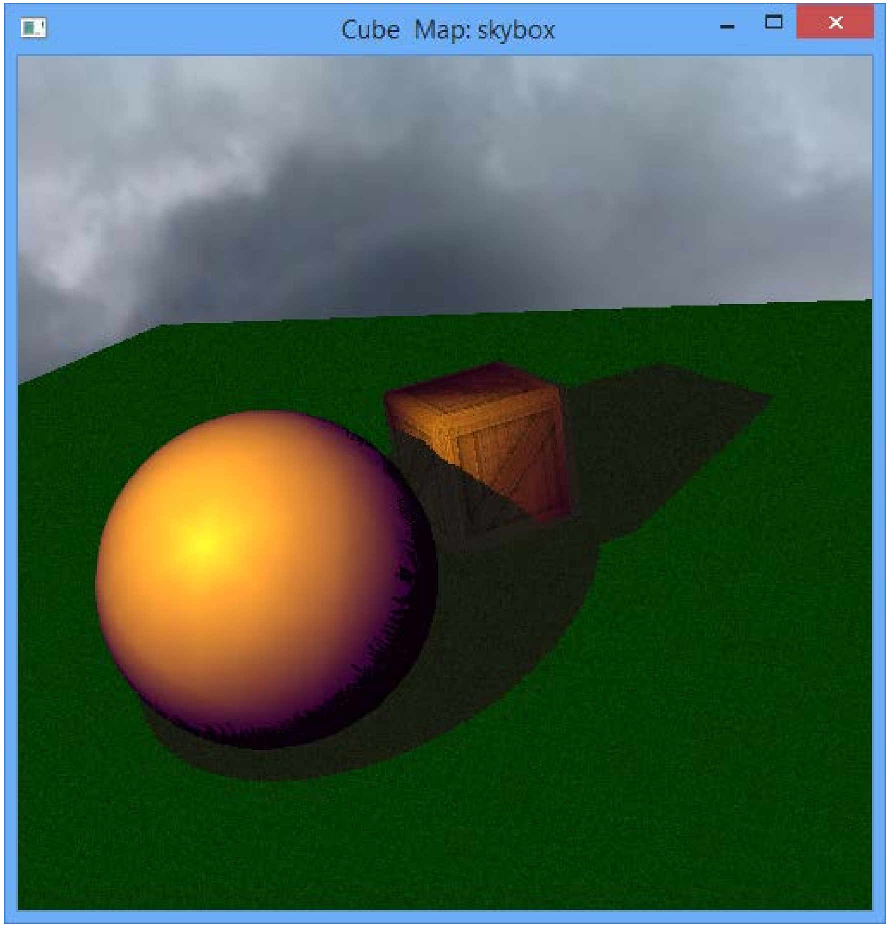


Shadow Mapping Issues



- Notice stuff on the sphere
  - Due to texture mapping issues
- Solutions:
  1. Smoothing/Anti-Aliasing
  2. Use larger texture
  3. Have more polygons to make surface
- See ideas at
  - http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/
  - http://www.fabiensanglard.net/shadowmapping/index.php (some old style stuff, but conceptually good)

Shadow Mapping



Shadow Mapping

- You can do point lights like we do environment mapping
  - Make a shadow map in each of the 6 directions (with 90 degree FOV)
  - Use these to cast shadows

Additional Notes…

- The approach to use just a regular 2D texture sampler has been replaced in modern versions with `sampler2DShadow` textures and the `textureProj` function
    - Chapter 7 of the OpenGL book talks about this
- But for our purpose of compatibility, we’ll stick with the method proposed.

More Notes

- If you have several light sources you can just:
  - Make a shadow map for each
    - So need multiple textures and one framebuffer for each
  - Pass each of these into the fragment shader for your comparisons