Blending, Anti Aliasing, Backface Culling

MEDIA DESIGN SCHOOL
GAME DEV

Blending

Blending

- Once an incoming fragment has passed all of the enabled fragment tests, it can be combined with the current contents of the color buffer in one of several ways.
- The default, is to overwrite the existing values
- Alternatively, you might want to combine the color present in the framebuffer with the incoming fragment color---a process called blending

Blending

- Blending is associated with the fragment's *alpha value.*
- Alpha is the fourth color component, and all colors in OpenGL have an alpha value
- it's a measure of translucency, and is what's used when you want to simulate translucent objects, like colored
- Glass or Water Surface for example

Blending

- Where color of a translucent object is a combination of that object's color with the colors of all the objects you see behind it
- OpenGL to do something useful with alpha, the pipeline needs more information than the current primitive's color (which is the color output from the fragment shader)
- It needs to know what color is already present for that pixel in the framebuffer

Blending

- In basic blending mode, the incoming fragment's color is linearly combined with the current pixel's color.
- *coefficients* control the contributions of each term
- those coefficients are called the *source-* and *destination-blending factors*

Blending

Final Colour = ($SrRs + DrRd$, //red

$SgGs + DgGd$, //green

$SbBs + DbBd$, //blue

$SaAs + DaAd$) //alpha

- (Sr, Sg, Sb, Sa) represent the source-blending **factors**. (Dr,Dg,Db,Da) represent the destination factors.
- (Rs,Gs, Bs,As) and (Rd,Gd, Bd,Ad) represent the **colors** of the source fragment and destinatio pixel respectively

Blending

- Controlling
  - Blending factors and
  - Blending Equation.

Blending

- Blending Factors
- You may call **glBlendFunc()** and choose two blending factors: the first factor for the source RGBA and the second for the destination RGBA
- void **glBlendFunc**(GLenum *srcfactor*, GLenum *destfactor*);

Blending

Constant	RGB Blend Factor	Alpha Blend Factor
GL_ZERO	$(0, 0, 0)$	0
GL_ONE	$(1, 1, 1)$	1
GL_SRC_COLOR	(R_s, G_s, B_s)	A_s
GL_ONE_MINUS_SRC_COLOR	$(1, 1, 1) - (R_s, G_s, B_s)$	$1 - A_s$
GL_DST_COLOR	(R_d, G_d, B_d)	A_d
GL_ONE_MINUS_DST_COLOR	$(1, 1, 1) - (R_d, G_d, B_d)$	$1 - A_d$
GL_SRC_ALPHA	(A_s, A_s, A_s)	A_s
GL_ONE_MINUS_SRC_ALPHA	$(1, 1, 1) - (A_s, A_s, A_s)$	$1 - A_s$
GL_DST_ALPHA	(A_d, A_d, A_d)	A_d

MEDIA DESIGN SCHOOL GAME DEV

Blending

- Blending Equation
- With standard blending, colors in the framebuffer are combined (using addition) with incoming fragment colors to produce the new framebuffer color.
- void **glBlendEquation**(GLenum *mode*);

Blending

Blending Mode Parameter	Mathematical Operation
GL_FUNC_ADD	$C_s S + C_d D$
GL_FUNC_SUBTRACT	$C_s S - C_d D$
GL_FUNC_REVERSE_SUBTRACT	$C_d D - C_s S$
GL_MIN	$\min(C_s S, C_d D)$
GL_MAX	$\max(C_s S, C_d D)$

- GL_FUNC_ADD by default

Addition and Subtraction

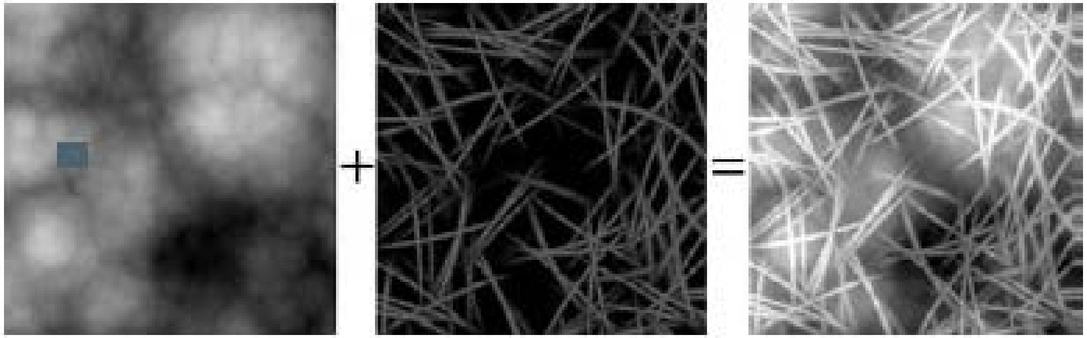


Figure 8.2: Adding source and destination color. Adding creates a brighter image since color is being added.

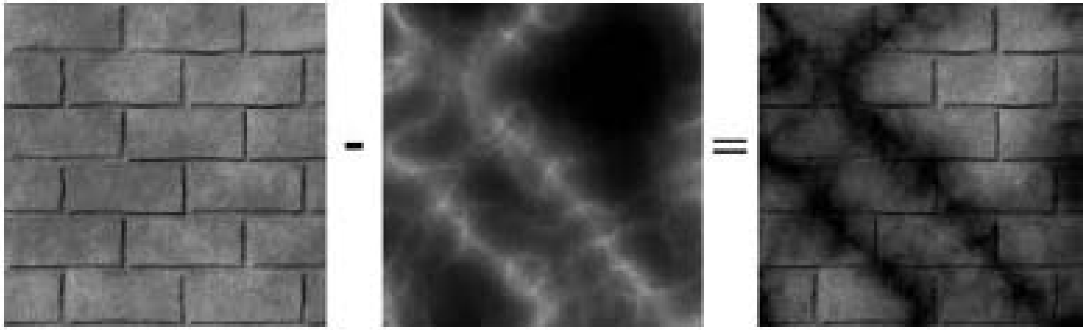


Figure 8.3: Subtracting source color from destination color. Subtraction creates a darker image since color is being removed.

Multiply

$$C = C_{src} \otimes F_{src} \boxplus C_{dst} \otimes F_{dst}$$

$$C = C_{src} \otimes (0,0,0) + C_{dst} \otimes C_{src}$$

$$C = C_{dst} \otimes C_{src}$$

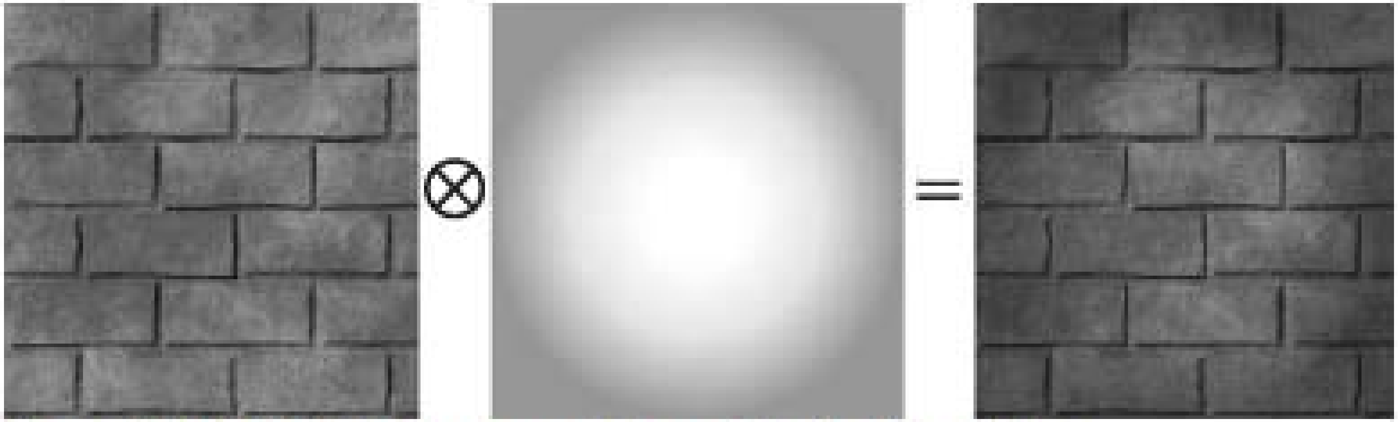


Figure 8.4: Multiplying source color and destination color.

Transparency

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

$$\mathbf{C} = \mathbf{C}_{src} \otimes (a_s, a_s, a_s) + \mathbf{C}_{dst} \otimes (1 - a_s, 1 - a_s, 1 - a_s)$$

$$\mathbf{C} = a_s \mathbf{C}_{src} + (1 - a_s) \mathbf{C}_{dst}$$

For example, suppose $a_s = 0.25$,

$$\mathbf{C} = a_s \mathbf{C}_{src} + (1 - a_s) \mathbf{C}_{dst}$$

$$\mathbf{C} = 0.25 \mathbf{C}_{src} + 0.75 \mathbf{C}_{dst}$$

Usage

- Render function

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
glBindVertexArray(0);

glDisable(GL_BLEND);
```

Notes

- While loading the texture using SOIL load the alpha channel data as well.
- unsigned char* image = SOIL_load_image(texFileName.c_str(), &width, &height, 0, **SOIL_LOAD_RGBA**);
- glTexImage2D(GL_TEXTURE_2D, 0, **GL_RGBA**, width, height, 0, **GL_RGBA**, GL_UNSIGNED_BYTE, image);

Notes

- The object with the transparency needs to be rendered at the end after rendering all other objects.
- JPEG doesn't support transparency. You'll need to stick to PNG or GIF.
- So the water texture was stored as a PNG for the effect to take place.

Anti Aliasing

Anti Aliasing

- Because the pixels on a monitor are not infinitely small, an arbitrary line cannot be represented perfectly on the computer
- stair-step" (aliasing) effect, which can occur when approximating a line by a matrix of pixels monitor

Anti Aliasing

- Shrinking the pixel sizes by increasing the monitor resolution can alleviate the problem significantly.
- When increasing the monitor resolution is not possible or not enough, we can apply antialiasing techniques.
- Supersampling
- Multisampling

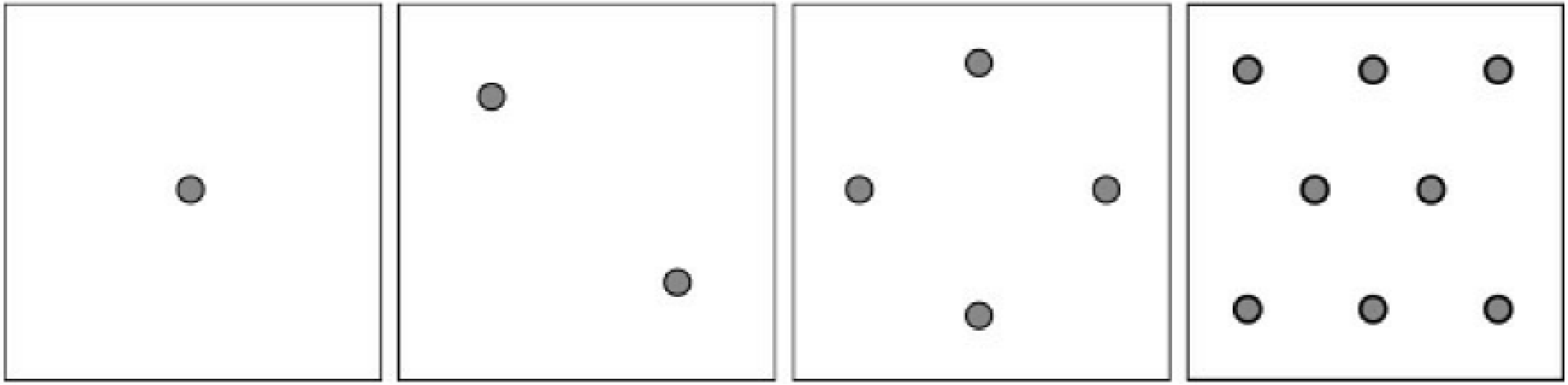
Anti Aliasing

- *supersampling*, works by making the back buffer and depth buffer 4X bigger than the screen resolution.
- The 3D scene is then rendered to the back buffer at this larger resolution
- when it comes time to present the back buffer to the screen, the back buffer is resolved (or downsampled) such that 4 pixel block colors are averaged together to get an averaged pixel color
- Supersampling is **expensive** because it increases the amount of pixel processing and memory by fourfold

Anti Aliasing

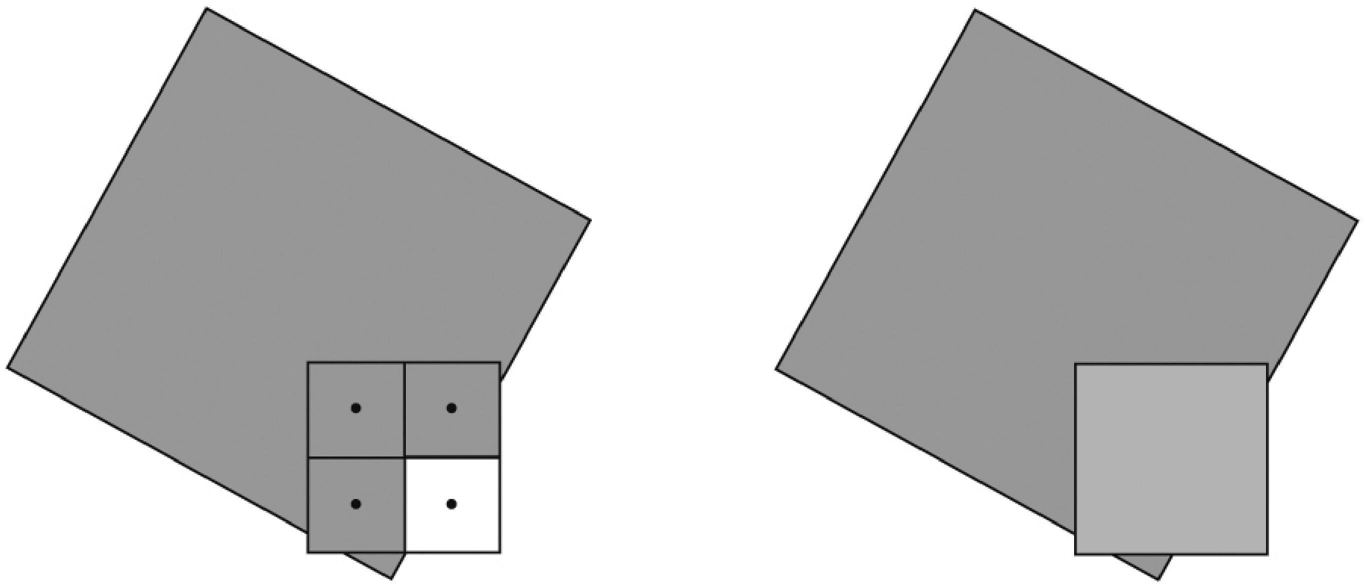
- To increase the sample rate of the image, store multiple samples for every pixel on the screen.
- This technique is known as multi-sample antialiasing (MSAA).
- Rather than sampling each primitive only once, OpenGL will sample the primitive at multiple locations within the pixel and, if any are hit, run your shader.

Anti Aliasing



MEDIA DESIGN SCHOOL GAME DEV

Anti Aliasing



(a) (b)

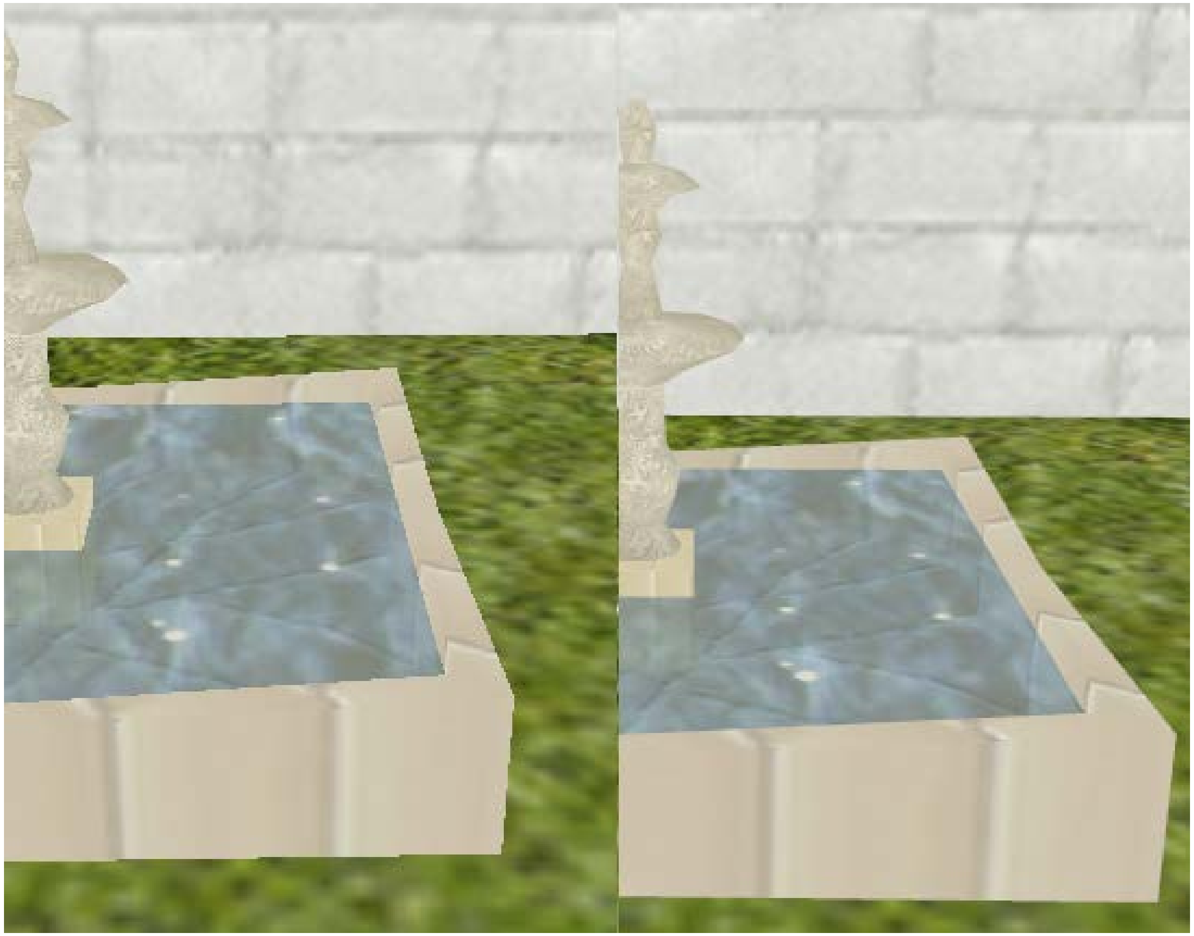
Usage

- In the init function at the start of the game.
- Initalize GLUT parameters in Main
- glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA| GLUT_MULTISAMPLE);
- Set multisample level

glutSetOption(GLUT_MULTISAMPLE, 8);

glEnable(GL_MULTISAMPLE);

Output



Backface Culling

Backface Culling

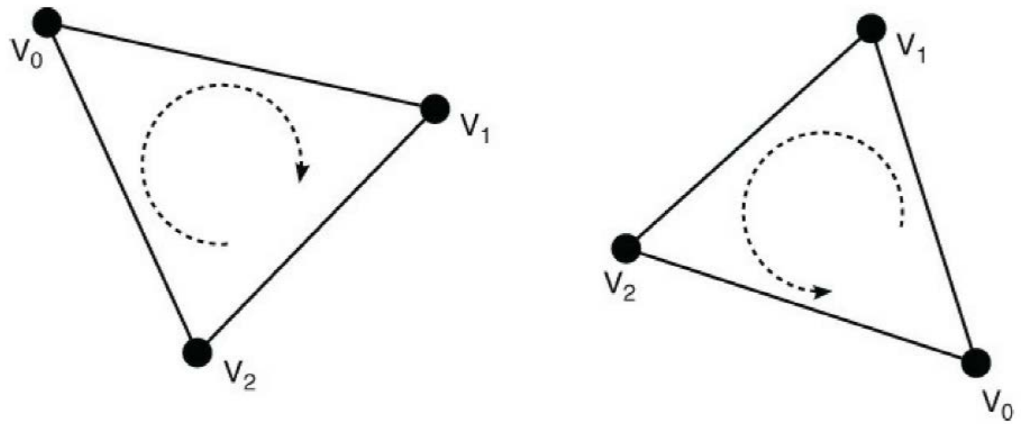
- The elimination of graphics primitives that would not be seen if rendered.
- Back-face culling eliminates the front or back face of a primitive so that the face isn’t drawn.
- Frustum culling eliminates whole objects that would fall outside the viewing frustum.

Backface Culling

- If the triangle faces toward the viewer, then it is considered to be *frontfacing*;
- otherwise, it is said to be *back-facing*
- It is very common to discard triangles that are back-facing because when an object is closed, any back-facing triangle will be hidden by another front-facing triangle.

Backface Culling

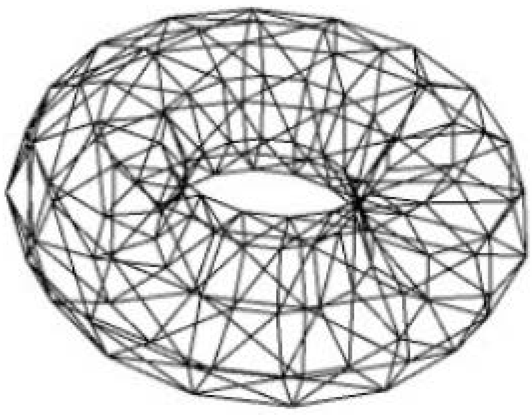
- Winding order determines if the face is front facing or back



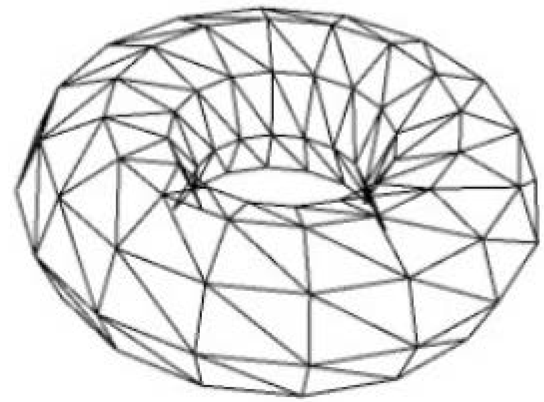
- glFrontFace(GL_CCW);
- Counterclock Wise by DEFAULT

Backface Culling

- Set the back faces to be culled.
- glCullFace(GL_BACK);



Torus drawn in wire-frame
<u>without</u> back face culling



Torus drawn in wire-frame
with back face culling

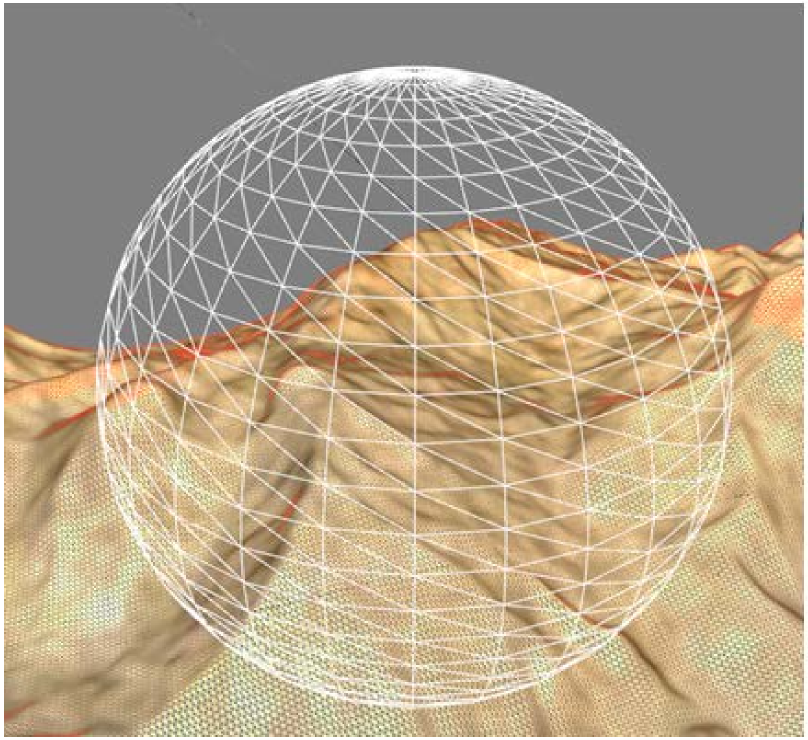
Backface Culling

- Enable face culling
- glEnable(GL_CULL_FACE);

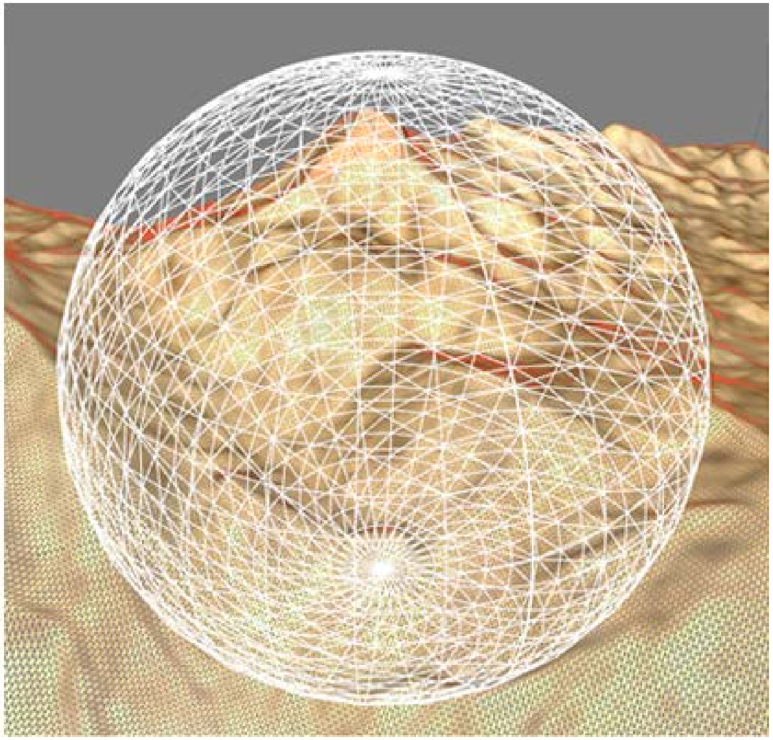
- Example

```
glEnable(GL_CULL_FACE);

glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
glBindVertexArray(0);

glDisable(GL_CULL_FACE);
```

Output



Enabled



Disabled

Exercises

- Add blending
- Animate the textures water surface
- Enable multisampling
- Enable back face culling for objects