VGP353 – Week 4

- Agenda:
  - Stencil-buffer refresher
  - Theory of shadow volumes
  - Generating shadow volume geometry

Types of Buffers

- OpenGL has three types of buffers:
  - Color buffers ...pixel values
  - Depth buffers ...distance to objects
  - Stencil buffers ...?

Stencil Buffer

- Extra per-pixel buffer containing integer values
  - Usually from 0 to 255 (for an 8-bit stencil buffer)
- Three basic operations:
  - Initialize it to a particular value (via `glClear()`)
  - Discard pixels based on the *Stencil Test*
  - Update it by drawing with a *Stencil Operation*
- You cannot texture from it
  - Unlike both color and depth

Clearing the Stencil Buffer

- Just like clearing color or depth:
  - `glClearStencil` sets the clear value
  - Pass `GL_STENCIL_BUFFER_BIT` to `glClear`
- If using depth *and* stencil, clear both together
  - Clearing both generally costs the same as clearing just one
  - Clearing one at a time can be expensive

Stencil Test

- Occurs after fragment shader, before depth test
- Set up via `glStencilFuncSeparate()`
  - Reference value (integer – 0, 127, 255...)
  - Comparison function
    - `GL_NEVER`, `GL_LESS`, `GL_EQUAL`, `GL_LEQUAL`, `GL_GREATER`, `GL_NOTEQUAL`, `GL_GEQUAL`, `GL_ALWAYS`
  - Mask bits (`~0` is effectively no mask)
- Performs bit-wise operations:
  - `(stencil & mask)` *`func`* `(ref & mask)`
- Fragments that fail the stencil test are discarded

Stencil Test

```
glStencilFuncSeparate(
  GLenum face,
  GLenum func,
  GLint ref,
  GLuint mask);
```

Stencil Test

```
glStencilFuncSeparate (
   GLenum face,
   GLenum func,
   GLint ref,
   GLuint mask);
```

Polygon facing selector: different operations for front and back facing polygons

Stencil Test

```
glStencilFuncSeparate (
  GLenum face,
  GLenum func,
  GLint ref,
  GLuint mask);
```

Polygon facing selector: different operations for front and back facing polygons

Comparison function

Stencil Test

```
glStencilFuncSeparate (
    GLenum face,
    GLenum func,
    GLint ref,
    GLuint mask);
```

Polygon facing selector: different operations for front and back facing polygons

Comparison function

Reference value used in comparison

Stencil Test

```
glStencilFuncSeparate (
   GLenum face,
   GLenum func,
   GLint ref,
   GLuint mask);
```

Polygon facing selector: different operations for front and back facing polygons

Comparison function

Reference value used in comparison

Bit-wise mask used on values before comparison

Stencil Test

```
glStencilFuncSeparate (
   GLenum face,
   GLenum func,
   GLint ref,
   GLuint mask);
```

Polygon facing selector: different operations for front and back facing polygons

Comparison function

Reference value used in comparison

Bit-wise mask used on values before comparison

- Passing `GL_FRONT_AND_BACK` for `face` acts like GL 1.x `glStencilFunc` function
  - Radeon r300 (e.g., Radeon 9800) needs front and back `ref` and `mask` to be the same

Updating the Stencil Buffer

- Eight possible stencil operations:
  - `GL_KEEP` – Keep existing value
  - `GL_ZERO` – Set value to zero
  - `GL_REPLACE` – Replace value with a reference value
  - `GL_INCR` – Increment value, clamp to max
    - `GL_INCR_WRAP` – Increment value, wrap to zero
  - `GL_DECR` – Decrement value, clamp to zero
    - `GL_DECR_WRAP` – Decrement value, wrap to max
  - `GL_INVERT` – Bitwise inversion of value
- Result is always masked with the stencil mask

Stencil Operation

- Stencil buffer values are modified per-fragment depending on the state of the fragment:
  - Fragment failed the stencil test
  - Fragment passed the stencil test but failed the depth test
  - Fragment passed the stencil test and passed the depth test
- You can specify a different operation for each case

Stencil Buffer

```
glStencilOpSeparate(
  GLenum face,
  GLenum sfail,
  GLenum dfail,
  GLenum dpass);
```

Stencil Buffer

```
glStencilOpSeparate(
   GLenum face,
   GLenum sfail,
   GLenum dfail,
   GLenum dpass);
```

Polygon facing selector: different operations for front and back facing polygons

Stencil Buffer

```
glStencilOpSeparate(
   GLenum face,
   GLenum sfail,
   GLenum dfail,
   GLenum dpass);
```

Polygon facing selector: different operations for front and back facing polygons

Operation when stencil test fails

Stencil Buffer

```
glStencilOpSeparate(
   GLenum face,
   GLenum sfail,
   GLenum dfail,
   GLenum dpass);
```

Polygon facing selector: different operations for front and back facing polygons

Operation when stencil test fails

Operation when stencil test passes but depth test fails

Stencil Buffer

```
glStencilOpSeparate(
  GLenum face,
  GLenum sfail,
  GLenum dfail,
  GLenum dpass);
```

Polygon facing selector: different operations for front and back facing polygons

Operation when stencil test fails

Operation when stencil test passes but depth test fails

Operation when stencil and depth tests pass

Stencil Buffer

```
glStencilOpSeparate(
   GLenum face,
   GLenum sfail,
   GLenum dfail,
   GLenum dpass);
```

Polygon facing selector: different operations for front and back facing polygons

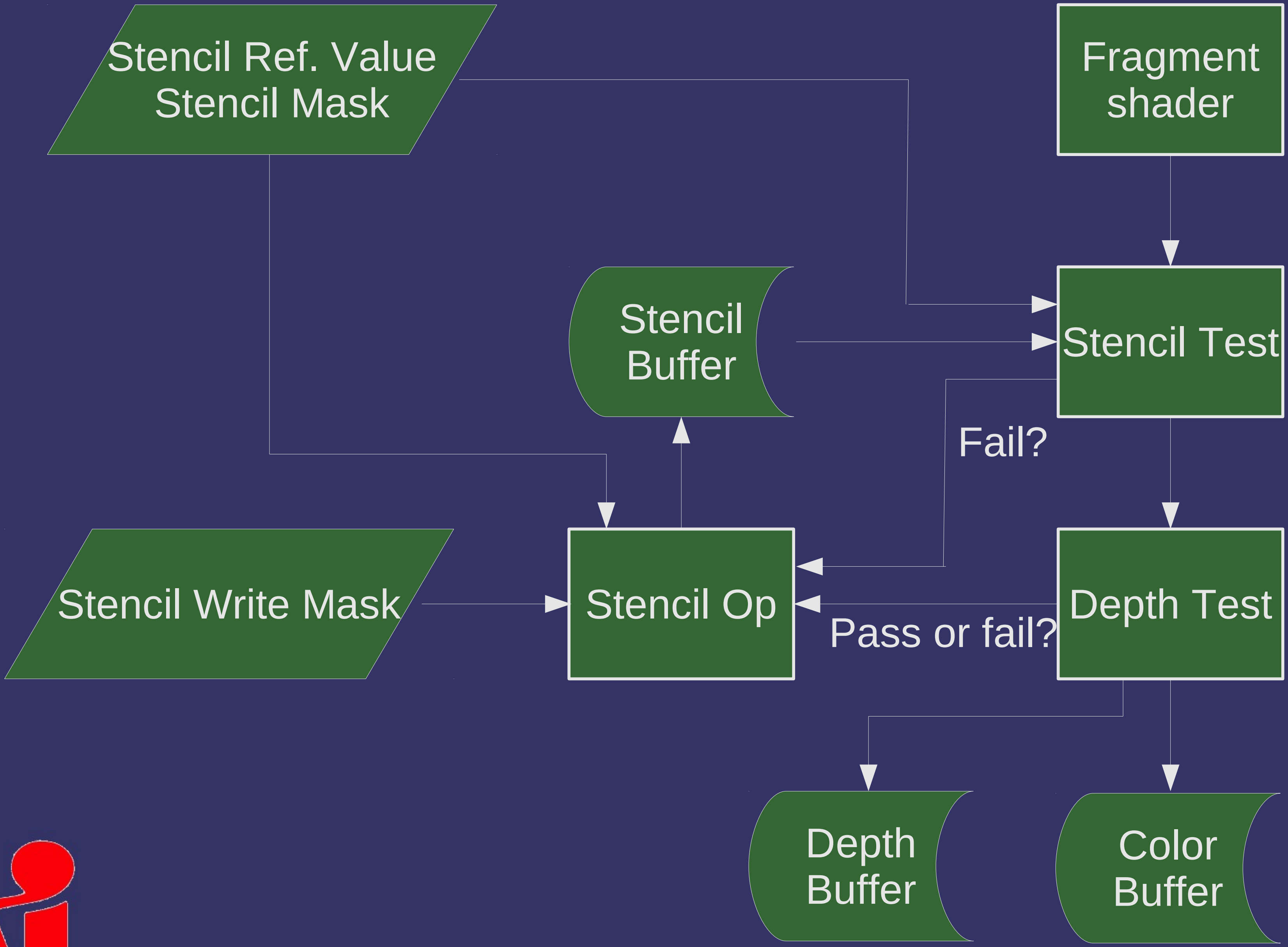
Operation when stencil test fails

Operation when stencil test passes but depth test fails

Operation when stencil and depth tests pass

- Passing `GL_FRONT_AND_BACK` for `face` acts like GL 1.x `glStencilOp` function

Stencil Buffer



Stencil Ref. Value
Stencil Mask

Fragment shader

Stencil Buffer

Stencil Test

Fail?

Stencil Write Mask

Stencil Op

Depth Test

Pass or fail?

Depth Buffer

Color Buffer

Stencil Buffer

- Writing of particular bits can be controlled with `glStencilMaskSeparate`
  - Passing `GL_FRONT_AND_BACK` for face parameter acts like GL 1.x `glStencilMask` function
  - Radeon r300 (e.g., Radeon 9800) needs front and back `mask` to be the same

Stencil Buffer – Example

```
glClearStencil(0);
glClear(GL_STENCIL_BUFFER_BIT);
glEnable(GL_STENCIL_TEST);

// Write 1 to stencil where polygon is drawn.
glStencilFuncSeparate(GL_FRONT_AND_BACK, GL_ALWAYS, 1, ~0);
glStencilOpSeparate(GL_FRONT_AND_BACK,
                    GL_KEEP, GL_KEEP, GL_REPLACE);
draw_some_polygon();

// Draw scene only where stencil buffer is 1.
glStencilFuncSeparate(GL_FRONT_AND_BACK, GL_EQUAL, 1, ~0);
glStencilOpSeparate(GL_FRONT_AND_BACK,
                    GL_KEEP, GL_KEEP, GL_KEEP);
draw_scene();
```

Stencil Buffer – Window System

- Stencil buffer is often stored interleaved with depth buffer
  - 8-bit stencil with 24-bit depth is most common
  - Others (e.g., 1-bit stencil w/15-bit depth) may exist
    - Very, *very* rare these days
- Must request a stencil buffer with your window
  - With SDL, this means setting the stencil size attribute to the minimum number of stencil bits required

```
SDL_GL_SetAttribute(SDL_GL_STENCIL_SIZE, 4);
```

Stencil Buffer – FBOs

- Stencil buffers can also be used with framebuffer objects
  - Create with `glRenderbufferStorage` and an internal type of `GL_STENCIL_INDEX`
    - Sized types are also available
    - There are *no* stencil textures
  - Attach to `GL_STENCIL_ATTACHMENT`

Stencil Buffer – FBO Example

```
glGenFramebuffers(1, &fb);
glGenTextures(2, tex_names);
glGenRenderbuffers(1, &stencil_rb);

// Setup color texture (mipmap)
glBindTexture(GL_TEXTURE_2D, tex_names[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, 512, 512, 0, GL_RGBA, GL_INT, NULL);
glGenerateMipmap(GL_TEXTURE_2D);

// Setup depth texture (not mipmap)
glBindTexture(GL_TEXTURE_2D, tex_names[1]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT24, 512, 512, 0,
             GL_DEPTH_COMPONENT, GL_UNSIGNED_INT, NULL);

// Setup stencil renderbuffer
glBindRenderbuffer(GL_RENDERBUFFER, stencil_rb);
glRenderbufferStorage(GL_RENDERBUFFER, GL_STENCIL_INDEX8, 512, 512);

glBindFramebuffer(GL_FRAMEBUFFER, fb);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                      GL_TEXTURE_2D, tex_names[0], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, tex_names[1], 0);
glFramebufferRenderBuffer(GL_FRAMEBUFFER, GL_STENCIL_ATTACHMENT,
                       GL_RENDERBUFFER, stencil_rb);
```

Stencil Buffer – FBOs

- If depth *and* stencil are required:
  - Create renderbuffer *<u>or</u>* texture with internal type of `GL_DEPTH_STENCIL`
    - The only sized type is `GL_DEPTH24_STENCIL8`
    - The `type` must be `GL_UNSIGNED_INT_24_8`
    - Treated as a depth texture for texturing
  - Bind same object to *<u>both</u>* the depth & stencil attachments
  - Requires OpenGL 3.0, `GL_ARB_framebuffer_objects`, or `GL_EXT_packed_depth_stencil`

Stencil Buffer – FBO Example

```
glGenFramebuffers(1, &fb);
glGenTextures(2, tex_names);

// Setup color texture (mipmap)
glBindTexture(GL_TEXTURE_2D, tex_names[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, 512, 512, 0, GL_RGBA, GL_INT, NULL);
glGenerateMipmap(GL_TEXTURE_2D);

// Setup depth_stencil texture (not mipmap)
glBindTexture(GL_TEXTURE_2D, tex_names[1]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8, 512, 512, 0,
             GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, NULL);

glBindFramebuffer(GL_FRAMEBUFFER_EXT, fb);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                      GL_TEXTURE_2D, tex_names[0], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, tex_names[1], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_STENCIL_ATTACHMENT,
                       GL_TEXTURE_2D, tex_names[1], 0);
```

Stencil Buffer – FBO Example

```
glGenFramebuffers(1, &fb);
glGenTextures(2, tex_names);

// Setup color texture (mipmap)
glBindTexture(GL_TEXTURE_2D, tex_names[0]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, 512, 512, 0, GL_RGBA, GL_INT, NULL);
glGenerateMipmap(GL_TEXTURE_2D);

// Setup depth_stencil texture (not mipmap)
glBindTexture(GL_TEXTURE_2D, tex_names[1]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8, 512, 512, 0,
             GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, NULL);

glBindFramebuffer(GL_FRAMEBUFFER_EXT, fb);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                      GL_TEXTURE_2D, tex_names[0], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, tex_names[1], 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_STENCIL_ATTACHMENT,
                       GL_TEXTURE_2D, tex_names[1], 0);
```

Same object attached both places

Shadow Volumes

Shadow Volumes

- Proposed by Frank Crow in 1977
  - Add new geometry to the scene that describes the volume occluded from the light source
  - Objects within the volume are in shadow, objects not within the volume are not
  - Sometimes called *Crow shadows* or *Crow shadow volumes*

Shadow Volumes

- Proposed by Frank Crow in 1977
  - Add new geometry to the scene that describes the volume occluded from the light source
  - Objects within the volume are in shadow, objects not within the volume are not
  - Sometimes called *Crow shadows* or *Crow shadow volumes*
- In 1991, Tim Heidmann showed how the stencil buffer can be used to apply these volumes to a scene
  - This adaptation often called *stencil volume shadows*

Shadow Volumes

- Basic algorithm:
  1. Render scene using only ambient light
  2. For each light in the scene:
     a. Using the depth information from the initial pass, construct a stencil with “holes” where there the light is not occluded.
        - Stencil will be 0 where the light is visible
     b. Render scene again with normal lighting. Use the stencil mask to only draw where the light is not occluded.
        - Configure stencil test to draw only where stencil = 0
  - Two common methods to create this stencil: z-pass and z-fail

Shadow Volumes

- Problems?

Shadow Volumes

- Problems?
  - ***Very*** fill-rate intensive
  - Calculating shadow volumes can be complex and time consuming
  - Difficult to extend to soft-shadows

Shadow Volumes

- Problems?
  - ***Very*** fill-rate intensive
  - Calculating shadow volumes can be complex and time consuming
  - Difficult to extend to soft-shadows
- Advantages?

Shadow Volumes

- Problems?
  - ***Very*** fill-rate intensive
  - Calculating shadow volumes can be complex and time consuming
  - Difficult to extend to soft-shadows
- Advantages?
  - Since everything is done in geometry-space instead of image-space, **no resampling aliasing artifacts!**
    - There is still aliasing, but not from resampling
  - No shadow acne

Shadow Volumes – Z-Pass

1. Disable depth and color writes
2. Configure stencil operation:
   - `GL_INCR_WRAP` on depth pass front-faces
   - `GL_DECR_WRAP` on depth pass back-faces
   - `GL_KEEP` for all other cases
3. Draw shadow volumes

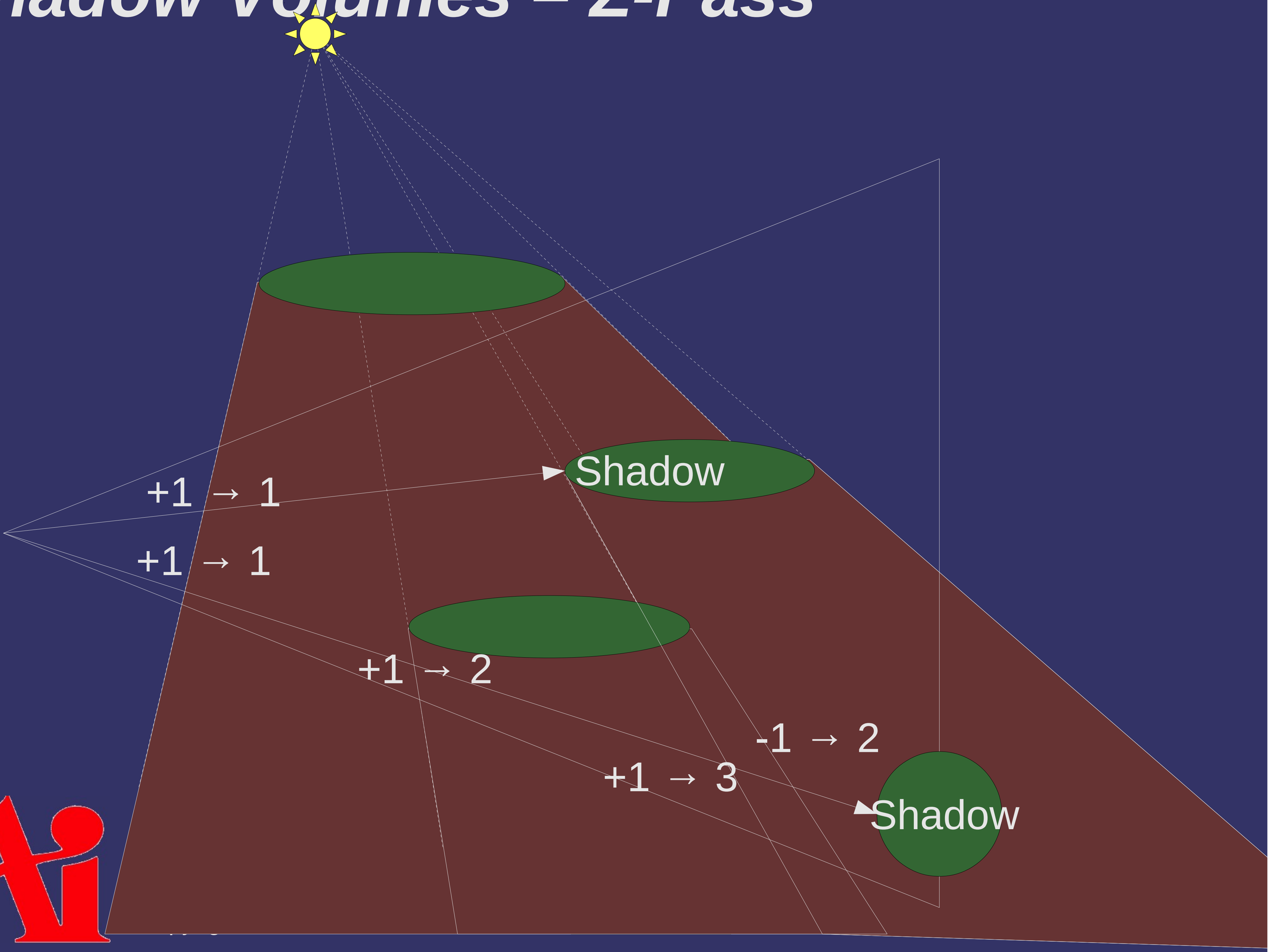
⇨ Why use `GL_INCR_WRAP` and `GL_DECR_WRAP` instead of `GL_INCR` and `GL_DECR`?

Shadow Volumes – Z-Pass

1. Disable depth and color writes
2. Configure stencil operation:
   - `GL_INCR_WRAP` on depth pass front-faces
   - `GL_DECR_WRAP` on depth pass back-faces
   - `GL_KEEP` for all other cases
3. Draw shadow volumes

- Why use `GL_INCR_WRAP` and `GL_DECR_WRAP` instead of `GL_INCR` and `GL_DECR`?
  - Otherwise, if there are more than 2^n increments before a decrement, the count will be wrong

Shadow Volumes – Z-Pass



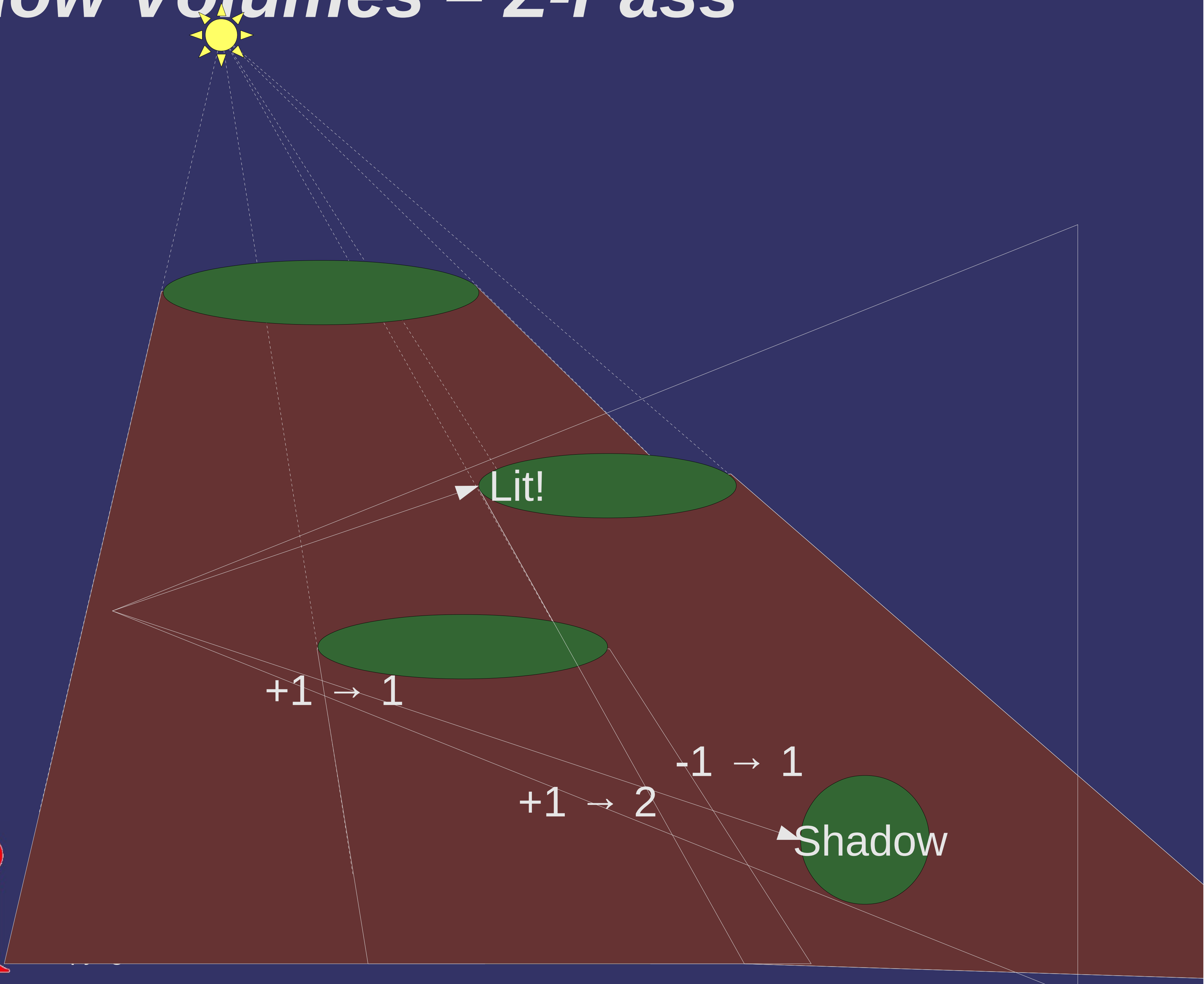
Shadow Volumes – Z-Pass

- Big problem with z-pass: What if the camera is *inside* a shadow volume?
  - Count is off by one for each volume the camera is in
  - Leads to areas being incorrectly labelled as illuminated or in-shadow

Shadow Volumes – Z-Pass

- Possible solutions:
  - Clear stencil buffer to +1 for each volume the camera is inside
    - Expensive to compute

Shadow Volumes – Z-Pass



Shadow Volumes – Z-Pass

- Another big problem with z-pass:
  - What if part of a shadow volume is clipped by the near plane?

Shadow Volumes – Z-Pass

- Possible solution:
  - Add a “cap” at the near plane for each volume the camera is inside
    - Expensive to compute
    - Robust implementation is difficult
- Maybe switch to a different method: z-fail

Shadow Volumes – Z-Fail

1. Disable depth and color writes
2. Configure stencil operation:
   - `GL_INCR_WRAP` on depth fail back-faces
   - `GL_DECR_WRAP` on depth fail front-faces
   - `GL_KEEP` for all other cases
3. Draw shadow volumes

- Method first *publicly* described by John Carmack while working on Doom 3
  - Sometimes called *Camack's reverse*

Shadow Volumes – Z-Fail

1. Disable depth and color writes
2. Configure stencil operation:
    - `GL_INCR_WRAP` on depth fail back-faces
    - `GL_DECR_WRAP` on depth fail front-faces
    - `GL_KEEP` for all other cases
3. Draw shadow volumes

Note: Depth test result and polygon facing are reversed compared to z-pass

Shadow Volumes – Z-Fail

- Advantages:
  - Correct counting when eye is inside shadow volumes
  - Doesn't miss intersections due to near-plane clipping
  - Avoids expensive workarounds for z-pass
- *Big* problems with z-fail:
  - Most geometry fails the depth test, so Z-fail can use orders of magnitude *more* fill rate
  - US Patent #6,384,822

Computing Shadow Volumes

- Shadow volume geometry is made of 3 types of polygons:
  - Front faces of the object (w.r.t. the light)
  - Quads from each silhouette edge (w.r.t. the light) projected to “infinity”
  - Back faces of the object (w.r.t. the light) projected to “infinity”

Computing Shadow Volumes

- For the sides...
  - Store two copies of each vertex: one with $w = 0$ and one with $w = 1$
  - Find silhouette edges on CPU by testing normals of polygons that share an edge with the light vector
    - An edge is a silhouette if one normal points towards the light and the other points away
  - For each silhouette edge, draw a quad using the four vertices on the edge
  - Use a special vertex shader that projects the $w = 1$ vertices away from the light to infinity

Extruding Sides a la Doom3

- Assembly vertex program from Doom3:

```
!!ARBvp1.0
TEMP R0;

# RO = OPOS - light, assumes light.w = 0
SUB   R0, vertex.position, program.env[4];

# RO -= OPOS.w * light
MAD   R0, R0.wwww, program.env[4], R0;

# normal transform
DP4   result.position.x, R0, state.matrix.mvp.row[0];
DP4   result.position.y, R0, state.matrix.mvp.row[1];
DP4   result.position.z, R0, state.matrix.mvp.row[2];
DP4   result.position.w, R0, state.matrix.mvp.row[3];
END
```

Extruding Sides a la Doom3

- Translated to GLSL...

```
uniform vec4 lightPos;  // assume w = 0
uniform mat4 mvp;
in vec4 vertex;

void main()
{
    // If w = 0: extrudedVertex = vertex - lightPos
    // If w = 1: extrudedVertex = vertex
    vec4 lightVec = vertex - lightPos;
    vec4 extrudedVertex = (lightPos * vertex.w)
        + lightVec;

    gl_Position = mvp * extrudedVertex;
}
```

Extruding Sides a la Doom3

- Where's the projection to infinity?!?
  - The extruded vertices have $w = 0$
  - In homogeneous coordinates, projection divides by w
  - Dividing by 0 pushes the vertex to infinity

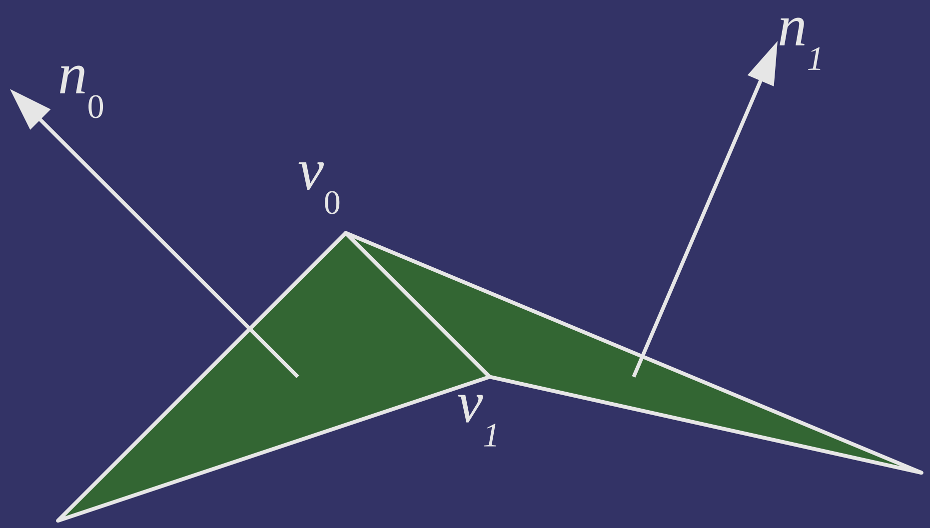
Extruding Sides a la Doom3

- Silhouette calculation on the CPU?
  - Slow!
  - Doesn't work well with nontrivial vertex transformations
    - Skinning, for example
  - Doesn't work well with hardware tessellation
  - Etc.

Extruding Sides on the GPU

- Do all of the extrusion in the vertex shader:
  - Store each vertex on an edge twice
    - One is paired with the normal of one triangle on the edge
    - One is paired with the normal of the other triangle
  - Draw a quad for **every** edge in the model
  - Extrude each vert if normal points away from the light
    - Silhouette edges will have one copy of the vertex extruded and the other not... creating a “real” quad
    - Non-silhouette edges will have both vertices extruded (or not) leaving a degenerate quad
- Similar to the fin extrusion in fins-and-shells fur

Extruding Sides on the GPU



Vertex data for shadow volume quad:

v0	n0
v1	n0
v1	n1
v0	n1

Shadow Volumes

- Advantages?
  - Shadow volume geometry is independent of light position and object orientation
  - Very little work done on the CPU per-frame
  - Static shadow volume data does not need to be re-uploaded to GPU every frame
- Disadvantages?
  - For static lights and geometry a *lot* of redundant work is done every frame
  - True shadow volumes only exist on the GPU, so we can't determine whether the camera is inside a shadow volume

References

Lengyel, Eric. “The Mechanics of Robust Stencil Shadows.” Gamasutra.com, October 11, 2002. http://www.gamasutra.com/view/feature/2942/the_mechanics_of_robust_stencil_.php

Yen Kwoon, Hun. “The Theory of Stencil Shadow Volumes.” GameDev.net, December 2, 2002. http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/the-theory-of-stencil-shadow-volumes-r1873

http://en.wikipedia.org/wiki/Shadow_volume

Next week...

- Back to shadow maps
  - Quantifying “resampling” aliasing issues
  - Fixing it!
- Quiz #2
  - Shadow map filtering
    - Reasons “regular” linear filtering can't be used
    - PCF
    - PCSS
  - Stencil volume shadows

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.