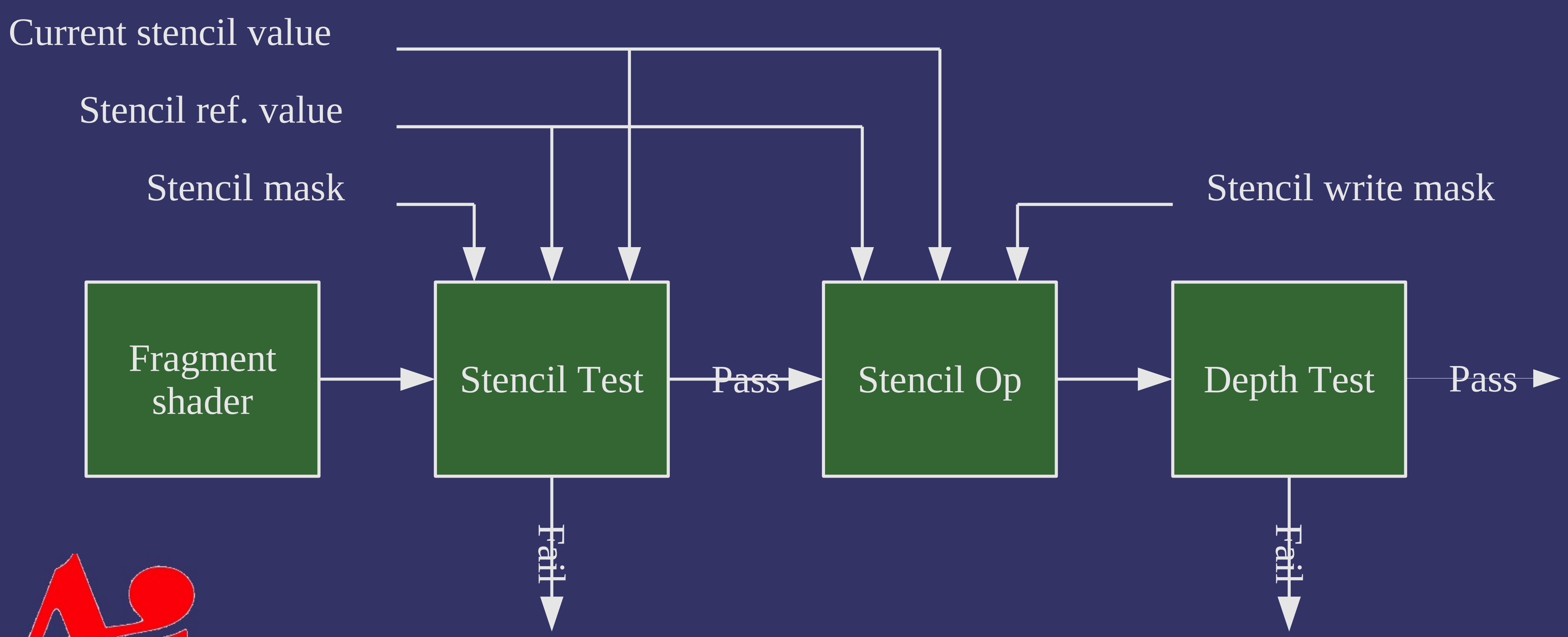
VGP353 – Week 5

- Agenda:
  - Quiz #2
  - Stencil-buffer refresher
  - Theory of shadow volumes
  - Generating shadow volume geometry

Stencil Buffer

- Extra per-pixel buffer containing integer values
  - Stencil test and stencil operation occur *after* per-fragment operations and *before* depth testing



Stencil Buffer

- Stencil function is one GL's usual comparators
  - `GL_NEVER`, `GL_LESS`, `GL_EQUAL`, `GL_LEQUAL`, `GL_GREATER`, `GL_NOTEQUAL`, `GL_GEQUAL`, `GL_ALWAYS`
  - Performs bit-wise operations of `(stencil & mask) func (ref & mask)`

Stencil Buffer

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

Stencil Buffer

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

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

Stencil Buffer

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

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

Comparison function

Stencil Buffer

```
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 Buffer

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

- face: Polygon facing selector: different operations for front and back facing polygons
- func: Comparison function
- ref: Reference value used in comparison
- mask: Bit-wise mask used on values before comparison

Stencil Buffer

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

- face: Polygon facing selector: different operations for front and back facing polygons
- func: Comparison function
- ref: Reference value used in comparison
- mask: 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

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

Stencil Operation

- Eight possible 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 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);
```

- face: Polygon facing selector: different operations for front and back facing polygons
- sfail: Operation when stencil test fails
- dfail: Operation when stencil test passes but depth test fails

Stencil Buffer

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

- face: Polygon facing selector: different operations for front and back facing polygons
- sfail: Operation when stencil test fails
- dfail: Operation when stencil test passes but depth test fails
- dpass: Operation when stencil and depth tests pass

Stencil Buffer

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

- `face`: Polygon facing selector: different operations for front and back facing polygons
- `sfail`: Operation when stencil test fails
- `dfail`: Operation when stencil test passes but depth test fails
- `dpass`: Operation when stencil and depth tests pass

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

Stencil Buffer

- Stencil buffer can also be cleared
  - `glClearStencil` sets the cleared value
  - Pass `GL_STENCIL_BUFFER_BIT` to `glClear`
  - If depth *and* stencil are used, always clear both together

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
  - Other combinations such as 1-bit stencil with 15-bit depth do 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 – FBOs

- If depth *and* stencil are required:
  - Create renderbuffer *or* texture with internal type of `GL_DEPTH_STENCIL`
    - One sized type of `GL_DEPTH24_STENCIL8`
    - `type` parameter must be `GL_UNSIGNED_INT_24_8`
    - Treated as a depth texture for texturing
  - Bind same object to both the depth and stencil attachments
  - Added with 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_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);
```

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

- 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 aliasing artifacts!!!**
  - No shadow acne either!

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

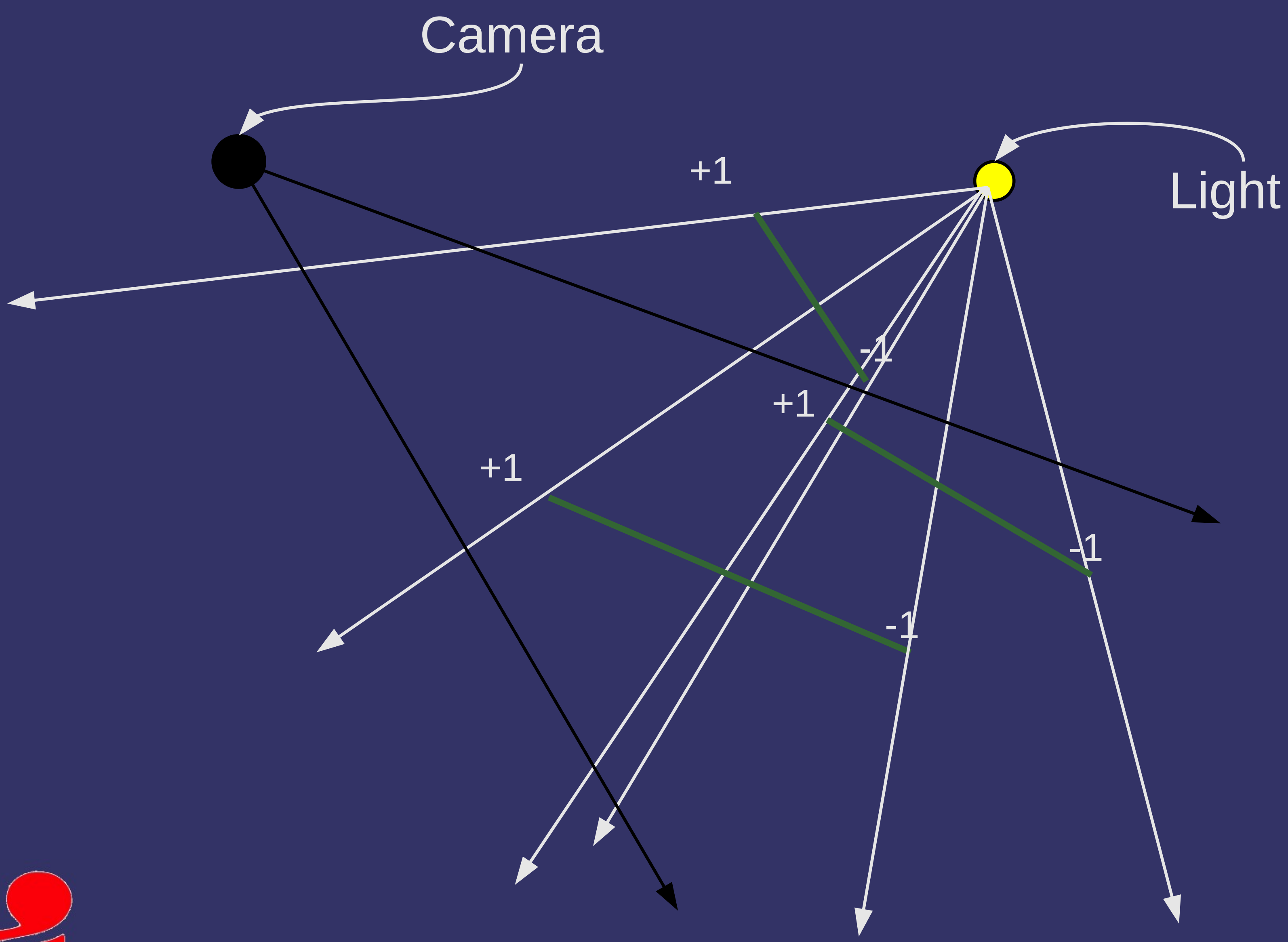
- Why use `GL_INCR_WRAP` and `GL_DECR_WRAP` instead of `GL_INCR` and `GL_DECR`?

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

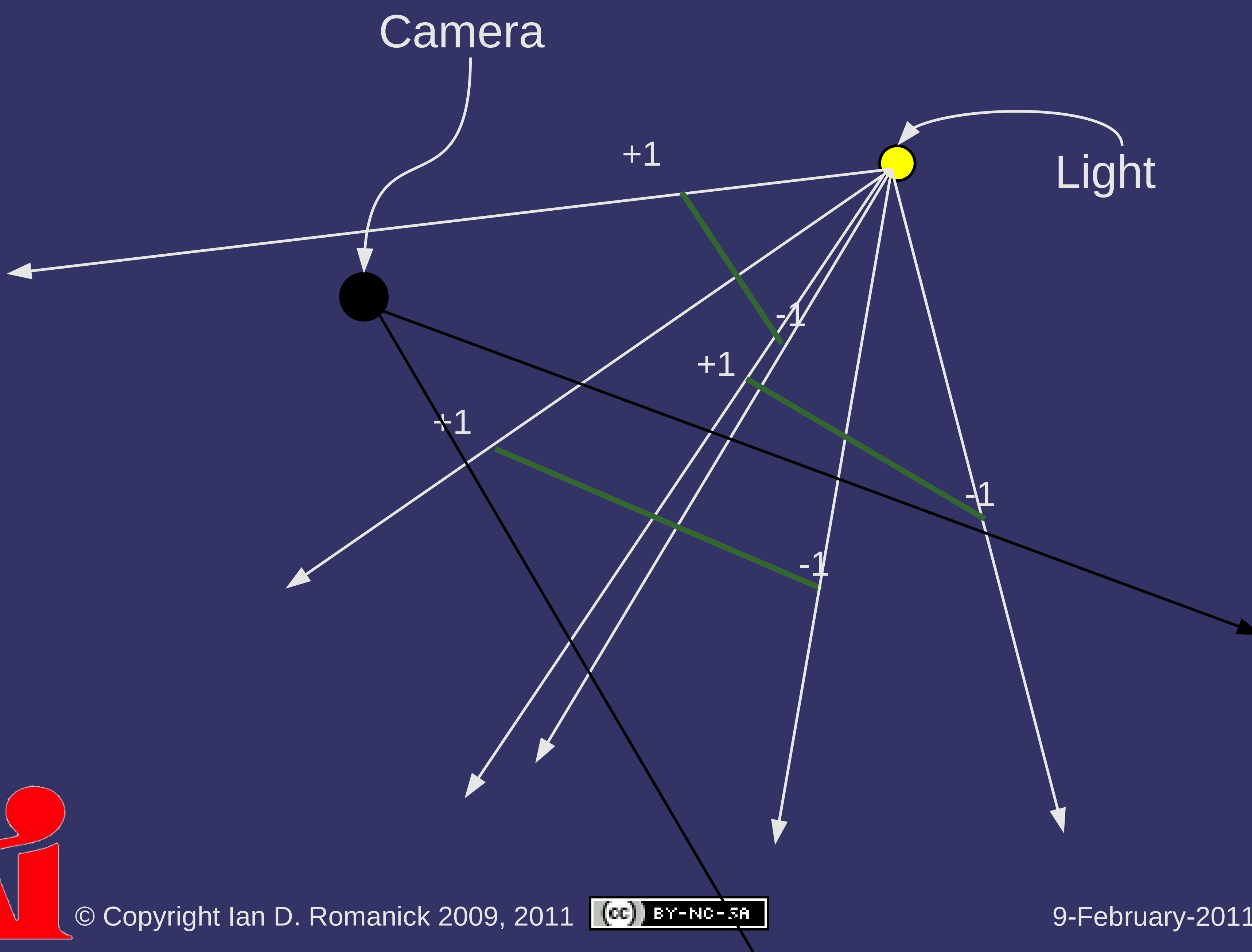
Shadow Volumes – Z-Pass



Shadow Volumes – Z-Pass

- Big problem with z-pass: What if the camera is *inside* a shadow volume?

Shadow Volumes – Z-Pass



Shadow Volumes – Z-Pass

- Big problem with z-pass: What if the camera is *inside* a shadow volume?
  - The count is too low!

Shadow Volumes – Z-Pass

- Big problem with z-pass: What if the camera is *inside* a shadow volume?
  - The count is too low!
- Possible solutions:
  - Clear stencil buffer to +1 for each volume the camera is inside
    - Expensive to compute
  - Add a "cap" at the near plane for each volume the camera is inside
    - Expensive to compute
  - Use 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
  - Often 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 that the depth test and the polygon facing are reversed compared to z-pass

Shadow Volumes – Z-Fail

- Big problems with z-fail:
  - Since more geometry fails the depth test than passes, this method can use orders of magnitude *more* fill rate
  - US Patent #6,384,822

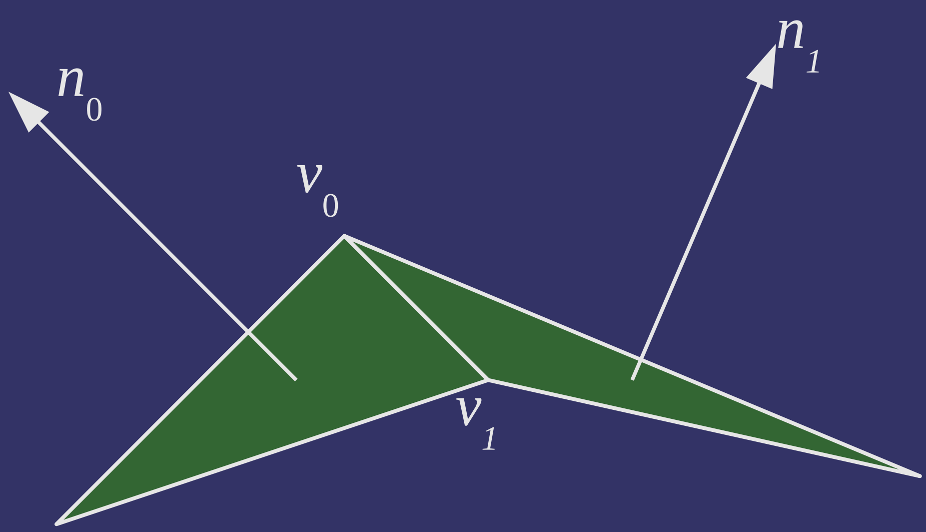
Shadow Volumes

- 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”

Shadow Volumes

- Front and back caps are trivial. What about the sides?
  - Add a degenerate quad at each edge of the model
  - Quad stores normals of one polygon with one vertex pair and normals of the other polygon with the other vertex pair
  - In vertex shader, test vertex normal against light. If normal points away from light, project to infinity
    - For silhouette edges one pair will be projected away and the other pair will not

Shadow Volumes



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

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

Shadow Volume Geometry

- Generating shadow volume geometry directly from raw vertex data is *hard*
  - Clearly some data structure is needed to make the work easier
- What features must this data structure have?

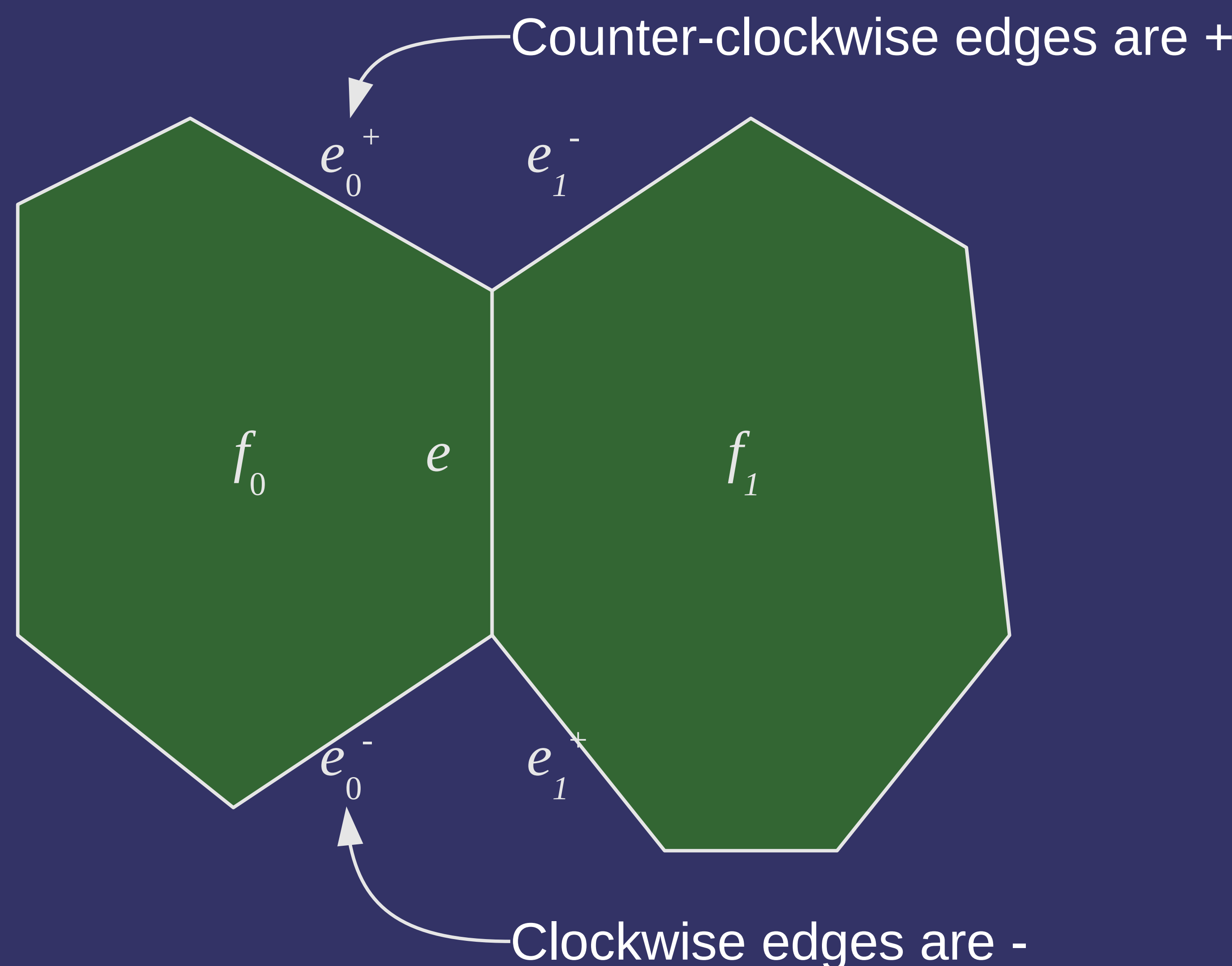
Shadow Volume Geometry

- Generating shadow volume geometry directly from raw vertex data is *hard*
  - Clearly some data structure is needed to make the work easier
- What features must this data structure have?
  - Iterate over each edge in the mesh *exactly once*
  - Access to each polygon sharing an edge
  - Access to neighboring edges in each polygon
    - This is so that normals can be calculated
- Does such a magical data structure exist?

Winged-Edge Mesh

- The *original* mesh structure to store connectivity information
- As the name implies, the focus is the *edge*
  - Each vertex stores a pointer to one of the edges "radiating" from it
  - Each polygon stores a pointer to one of its edges
  - Each edge has 8 pointers:
    - Pointers to each of its vertices (2)
    - Pointers to each of its polygons (2)
    - Pointers to each of its connecting edges (4)

Winged-Edge Mesh



Winged-Edge Mesh

- Desirable mesh representation properties:
  - Ease of manipulation: adding and removing data should not be too expensive
  - Scalability: May want to trade data size for performance per the needs of the application

Winged-Edge Mesh

- Desirable mesh representation properties:
  - Ease of manipulation: adding and removing data should not be too expensive
  - Scalability: May want to trade data size for performance per the needs of the application

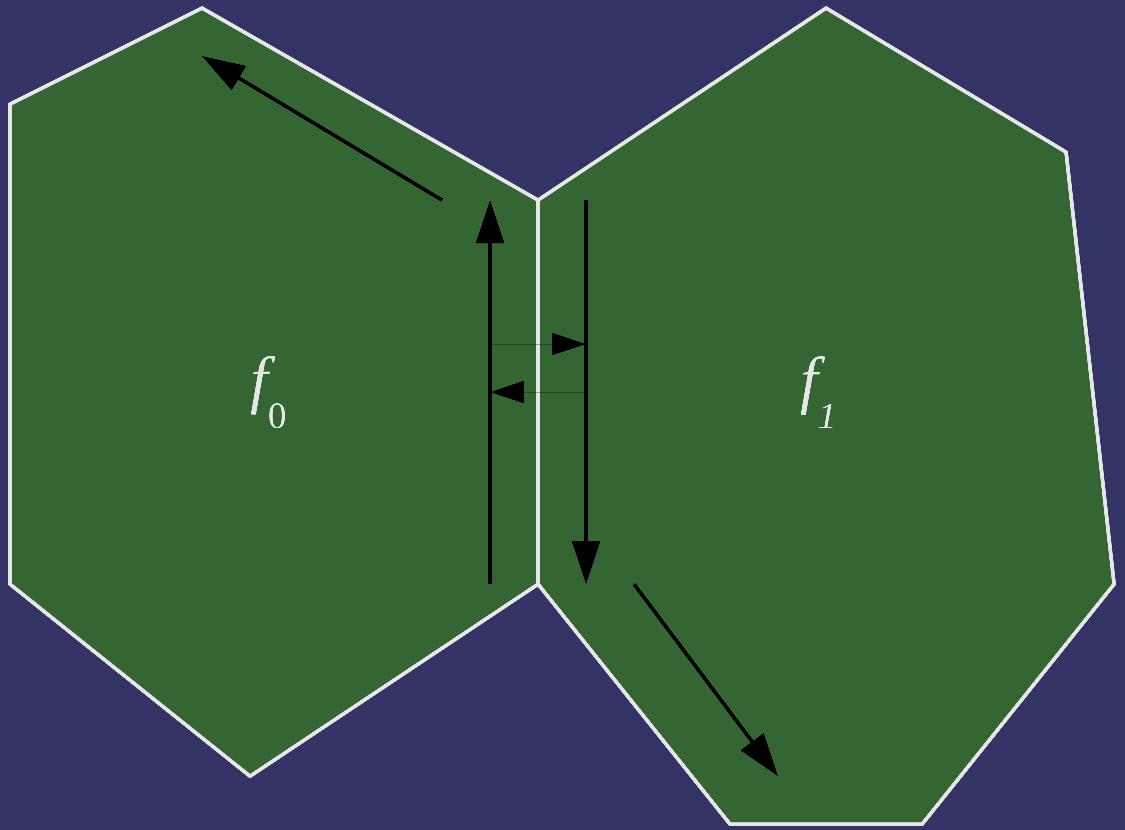
- Several common types of updates on WE meshes are *really* complicated to implement correctly

- Base winged-edge lacks the ability to iterate over the edges
- Base winged-edge has a *lot* of extra pointers that we will never use

Half-Edge Mesh

- Slight modification of winged-edge mesh:
  - Half-edge (HE) structures replace (full) edges
  - Each HE stores 4 pointers:
    - Pointer to starting vertex (1)
    - Pointer to polygon (1)
    - Pointer to counter-clockwise neighbor HE on the same polygon (1)
    - The “opposite” HE (1)
      - I call this the *sibling edge*
      - Other references call it *symmetric edge* or *pair edge*

Half-Edge Mesh



Half-Edge Mesh

```
struct half_edge {
    // Pointer to next counter-clockwise edge on same
    // polygon
    struct half_edge *next_ccw;

    // Pointer to matching edge on different polygon
    struct half_edge *sibling;

    // Pointer to the owning polygon
    struct polygon *p;

    // Pointer to next edge in global mesh edge list
    struct half_edge *next;

    // Pointer to starting vertex
    struct vertex *v;
};
```

Half-Edge Mesh

- If each HE only stores one vertex pointer, how do we get the other end?

Half-Edge Mesh

- If each HE only stores one vertex pointer, how do we get the other end?
  - The sibling edge stores a pointer to the other vertex
  - `e->v` and `e->sibling->v` make up the complete edge

Half-Edge Mesh

```
struct vertex {
    // Pointer an edge leaving this vertex
    struct half_edge *edge;

    // Pointer to position data for this vertex
    GLUvec4 *v;
};
```

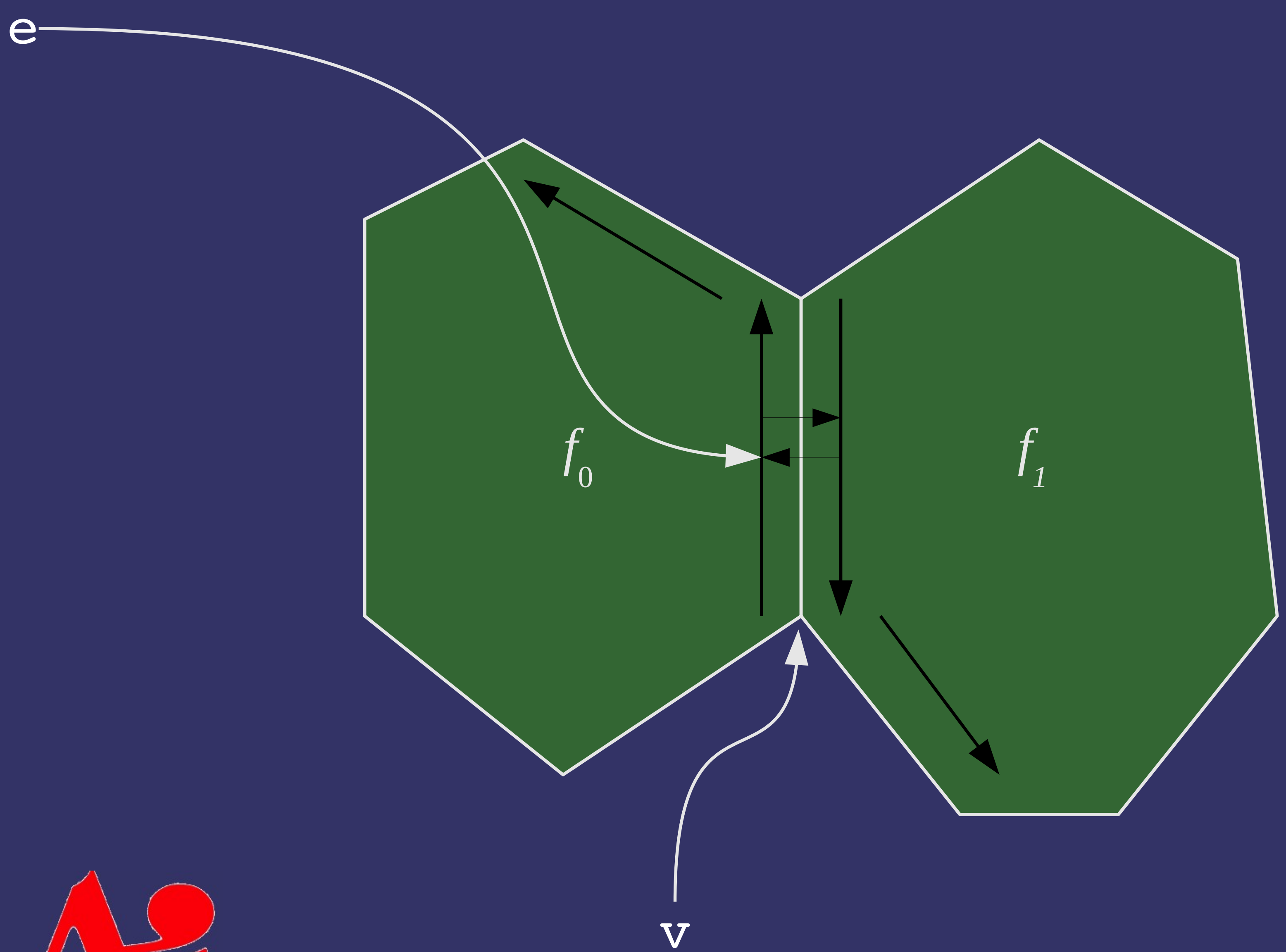
Half-Edge Mesh

- Given a vertex structure, how can we iterate all the edges that share that vertex?

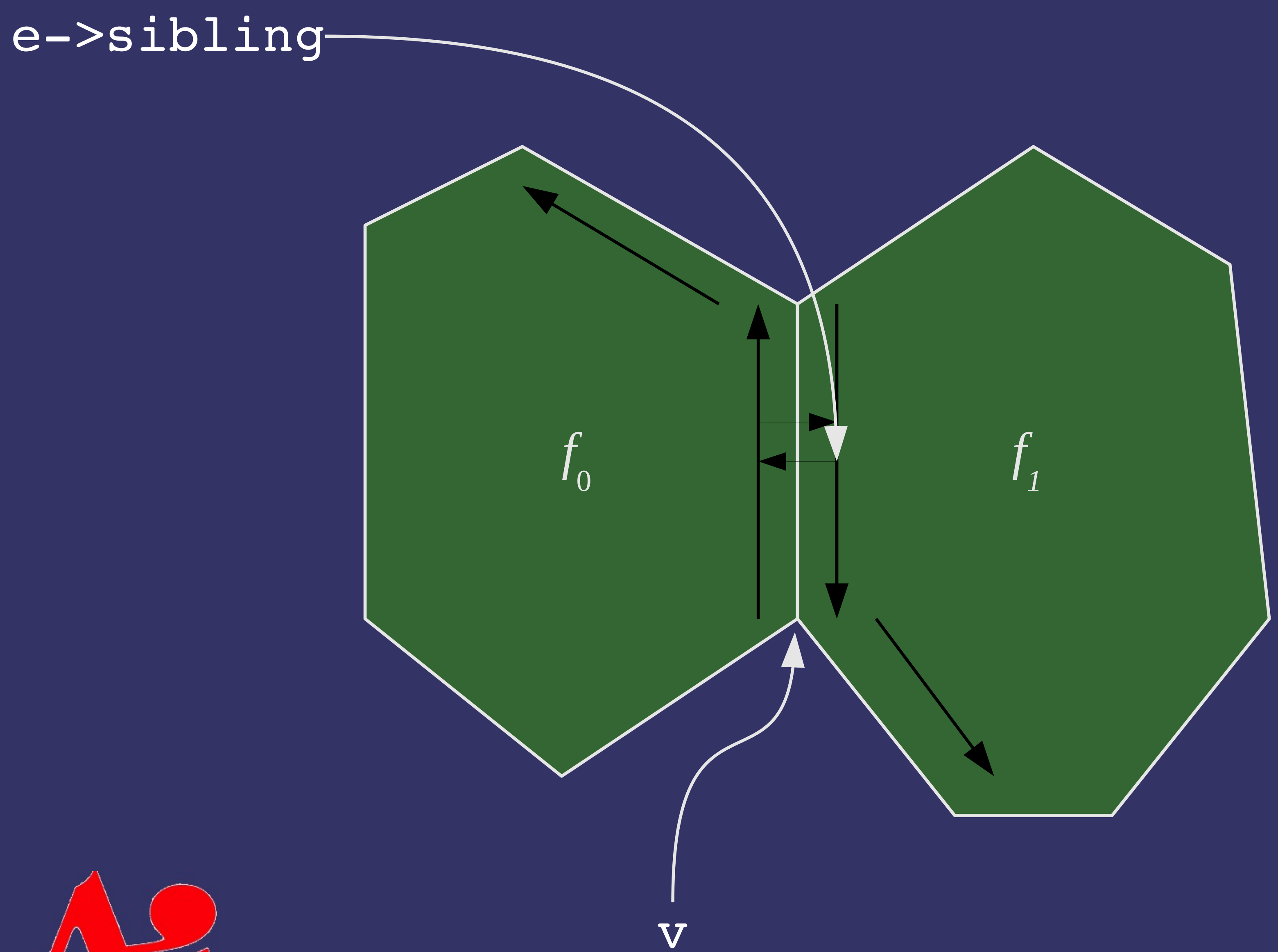
```
half_edge *e = v->edge;
do {
    // Do real work here.

    // Iterate to next edge
    e = e->sibling->next_ccw;
} while (e != v->edge);
```

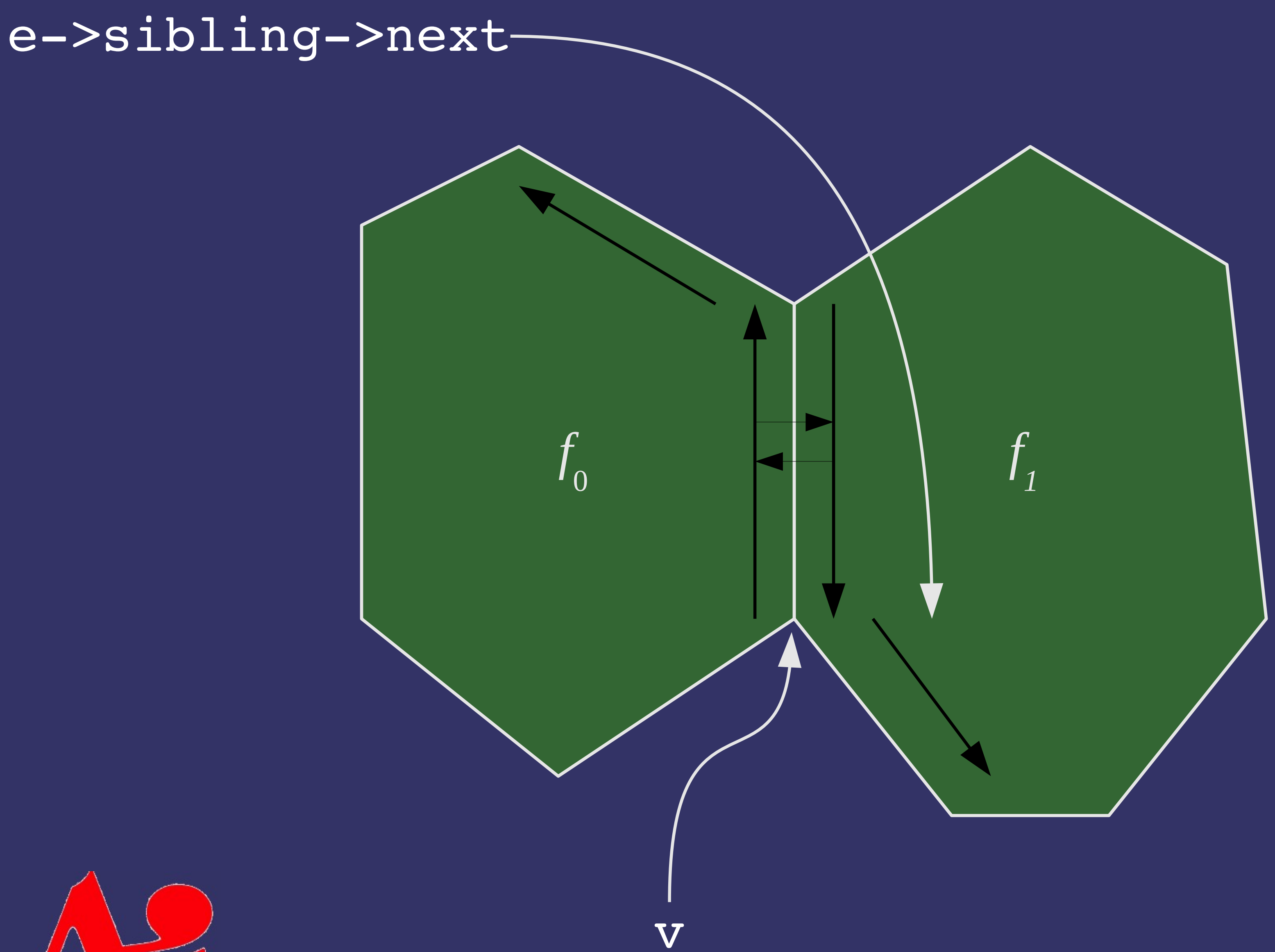
Half-Edge Mesh



Half-Edge Mesh

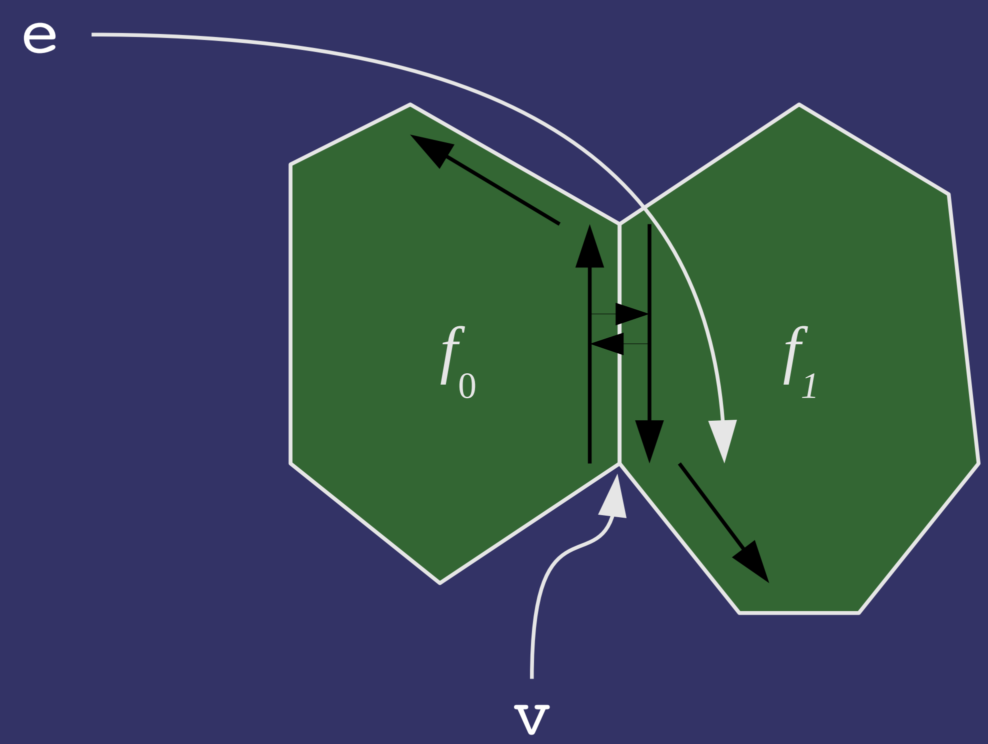


Half-Edge Mesh



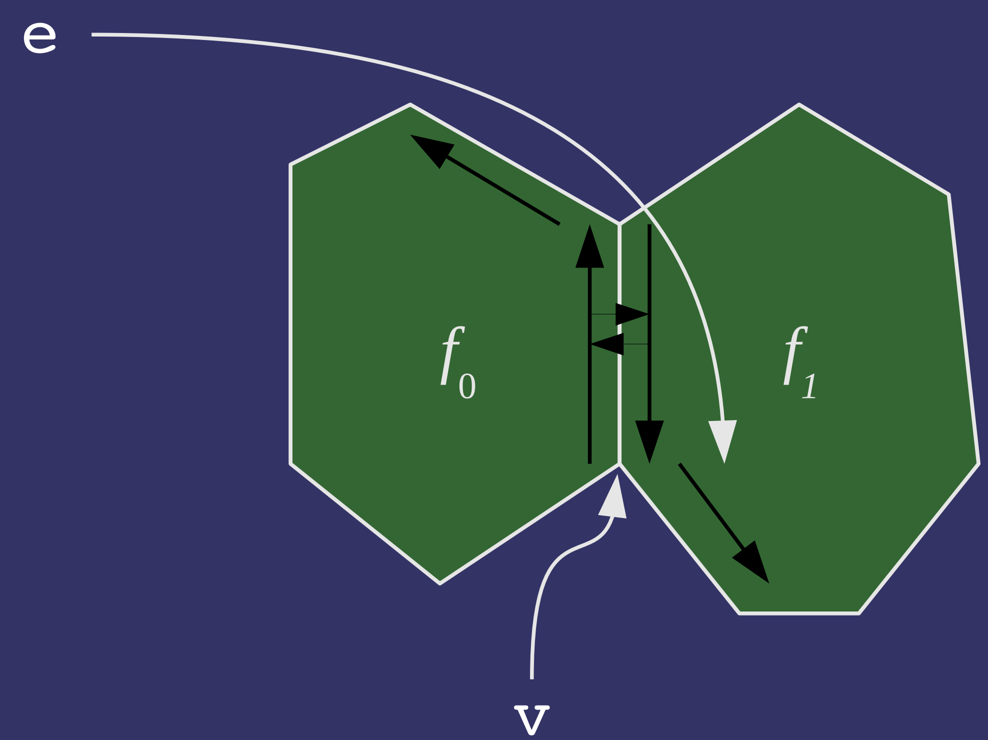
Half-Edge Mesh

- What's the problem?



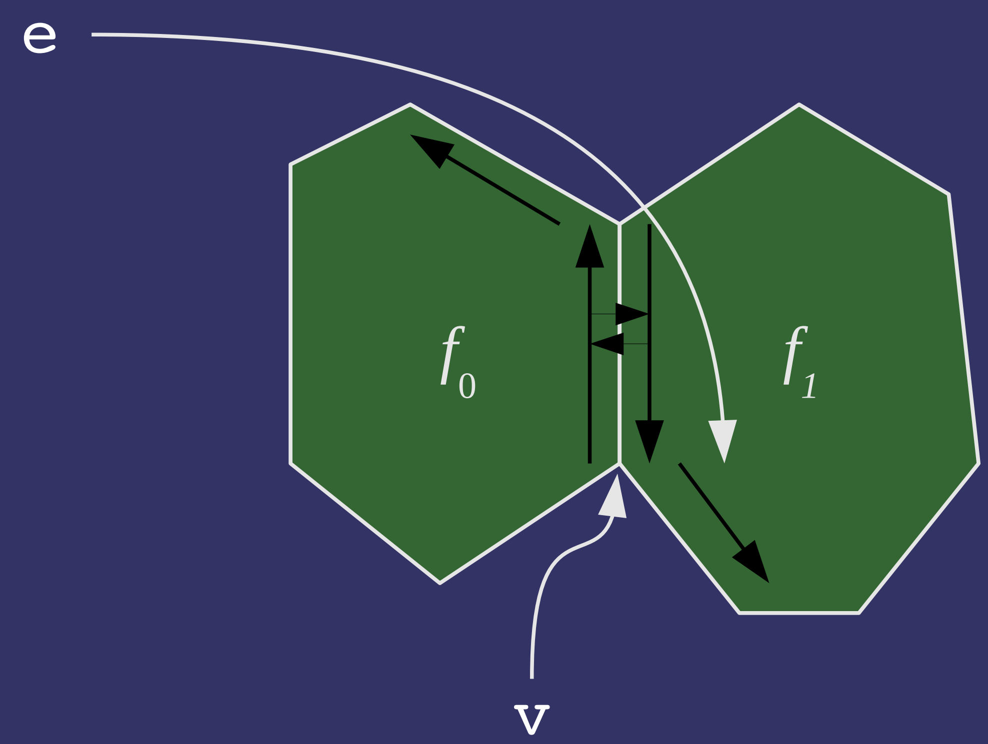
Half-Edge Mesh

- What's the problem?
  - The new e doesn't really have a sibling!
  - There are no pointers to follow to get the next edge



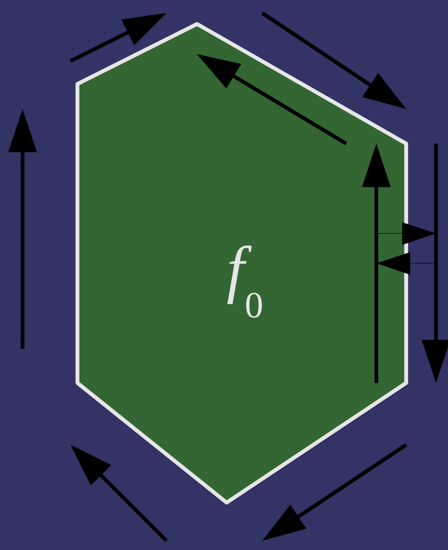
Half-Edge Mesh

- How can we add new edges to the mesh and prevent this problem?



Half-Edge Mesh

- How can we add new edges to the mesh and prevent this problem?
  - As new polygons are created, the sibling edges are linked in a “fake” CCW ring
    - The polygon pointers of these HEs is `NULL`
  - Adding *new* edges is a matter of updating all the linked lists



Half-Edge Mesh

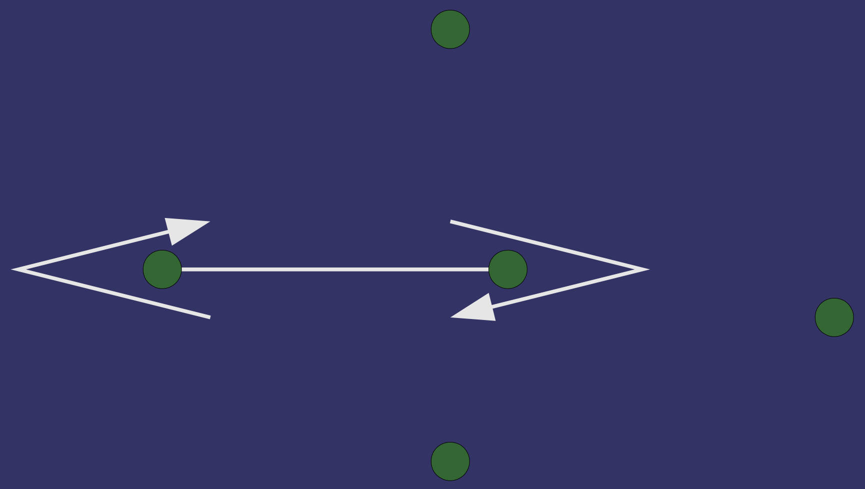
- To make the HE work, there are a few more primitives required
  - `create_edge(v0, v1)`: Create a new pair of HEs between `v0` and `v1`
  - `make_adjacent(a, b)`: Link `a` and `b` so that `a->next = b`
  - `add_polygon(edges, n)`: Create a new polygon from a list of existing edges

Half-Edge Mesh

- To create a *new* edge:
  - Allocate two HEs, link one to `v0` and the other to `v1`
  - Set both polygon pointers to `NULL`
  - Link both HEs as siblings
  - Link both HEs as each others `next_ccw`
    - Tricky! This makes the bootstrap case work and fixes other issues in `make_adjacent`
  - Insert each edge in the “gap” in the vertex's edge list
    - Some HE where:
      - `e->sibling->v == v`
      - `e->p == NULL`
      - `e->next_ccw->v == v`

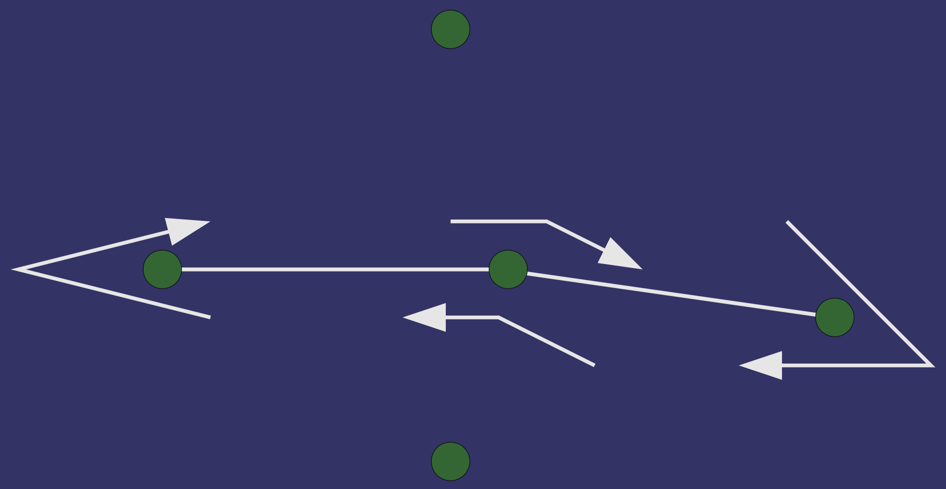
Half-Edge Mesh

- Edges can be added in arbitrary order...



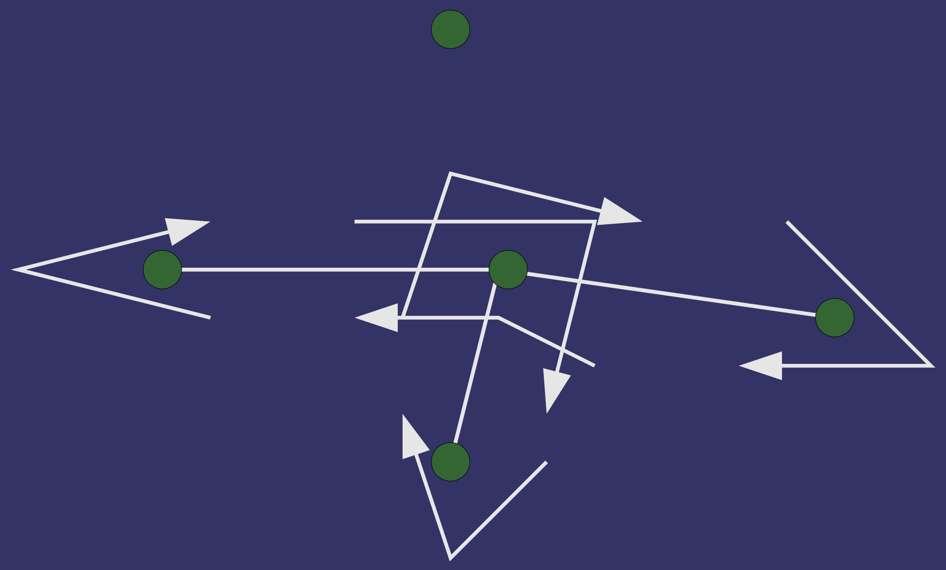
Half-Edge Mesh

- Edges can be added in arbitrary order...



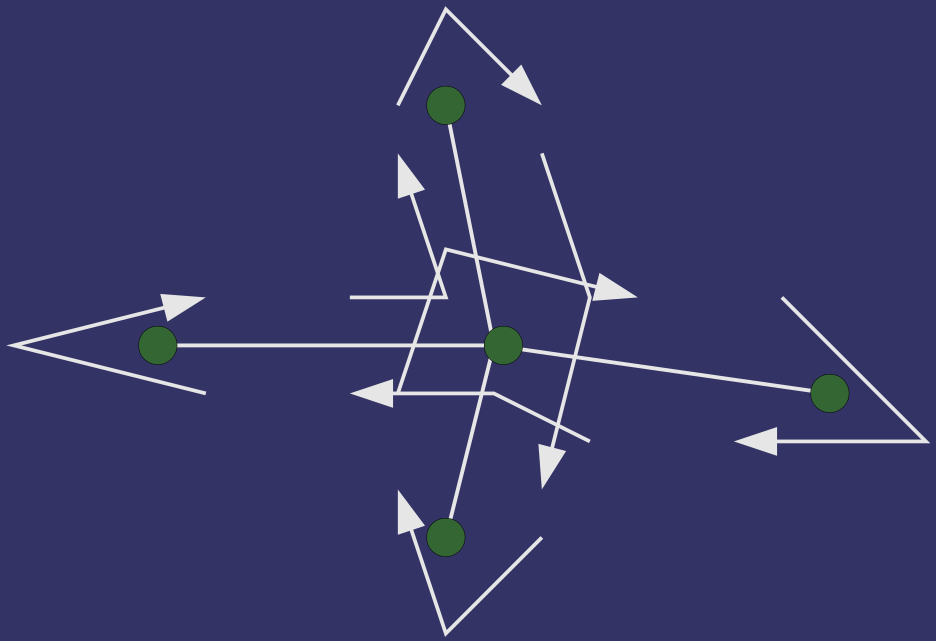
Half-Edge Mesh

- Edges can be added in arbitrary order...



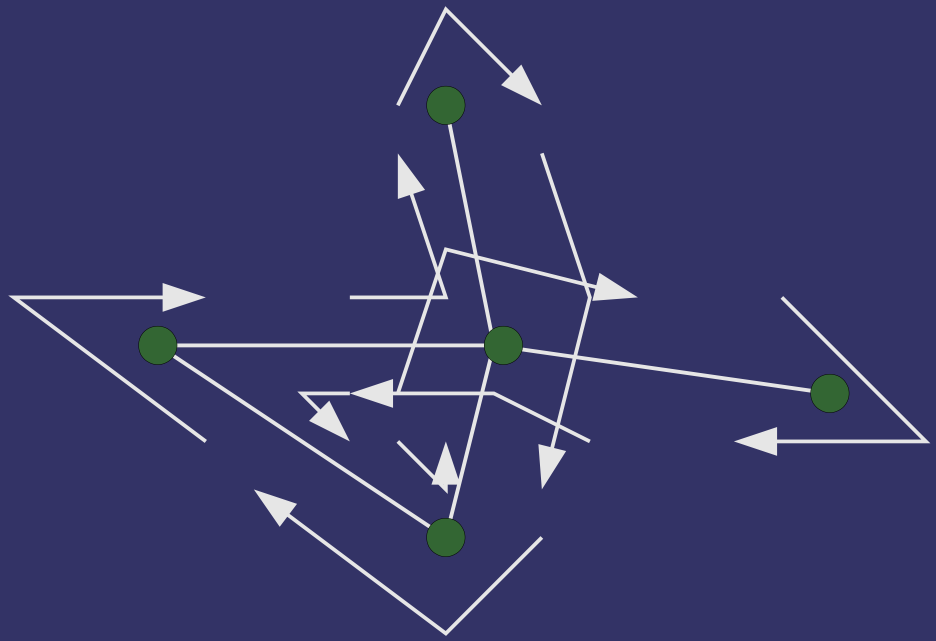
Half-Edge Mesh

- Edges can be added in arbitrary order...



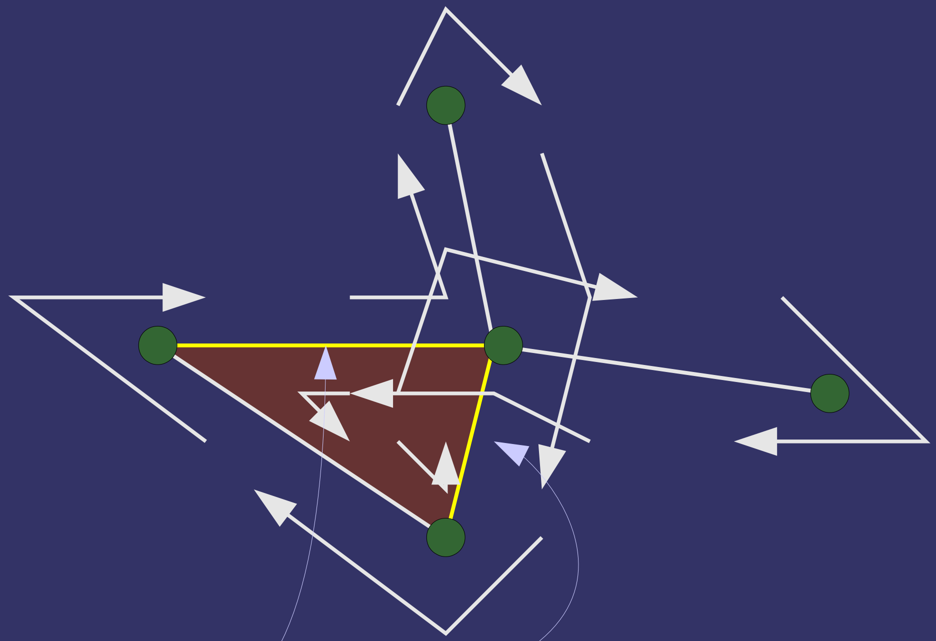
Half-Edge Mesh

- Edges can be added in arbitrary order...



Half-Edge Mesh

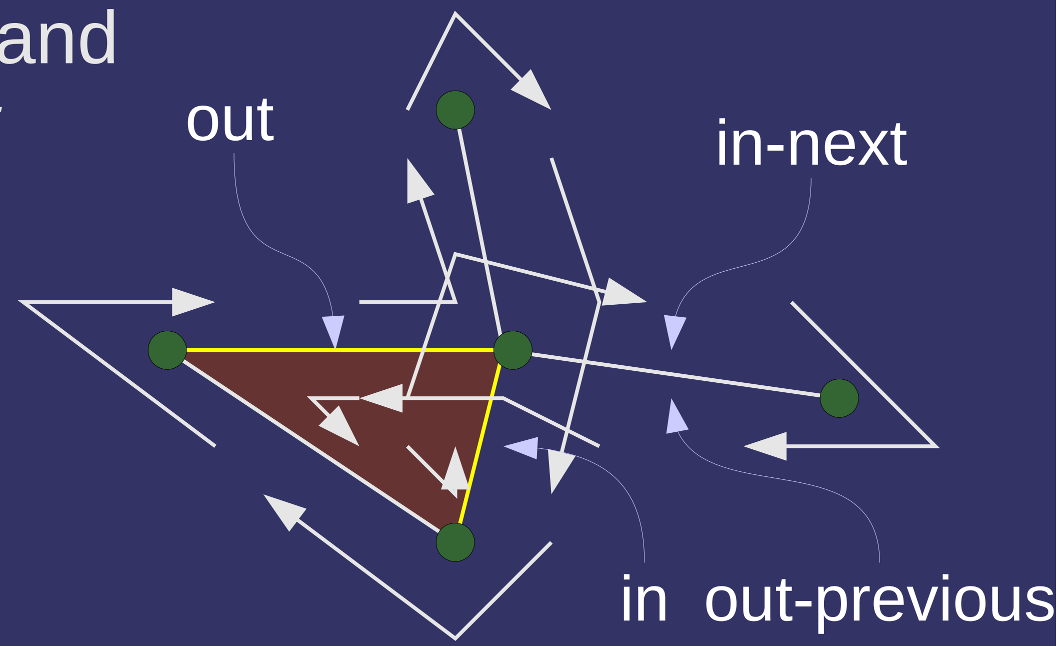
- Edges can be added in arbitrary order...
  - This causes problems when edges are formed into a polygon



These edges should be linked

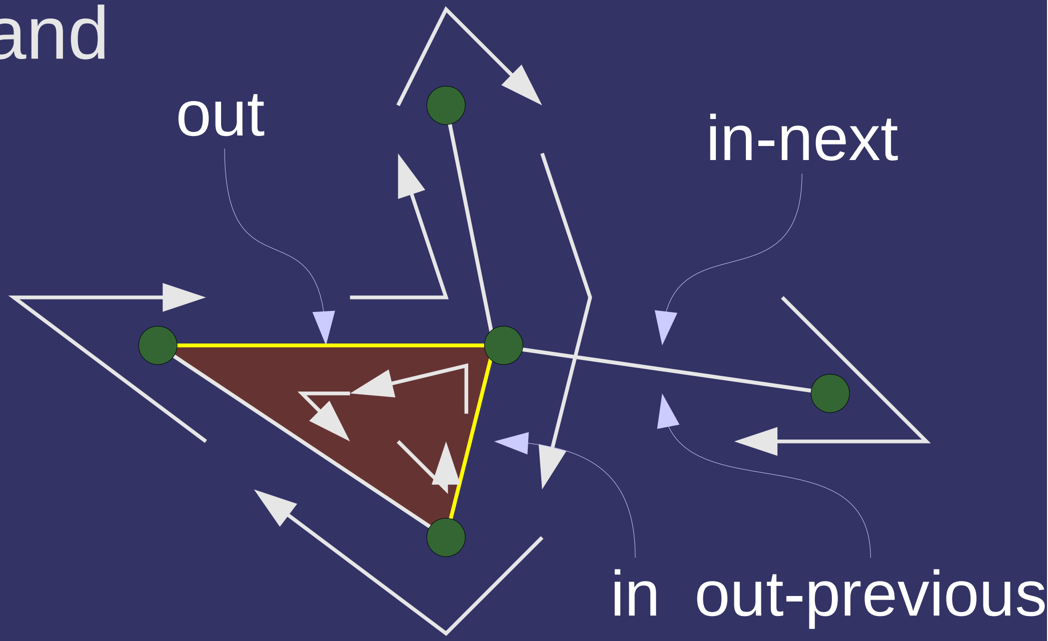
Half-Edge Mesh

- Relink the edges to create the correct relationships
  - Cut the links between *in* and *in-next*, and between *out* and *out-previous*



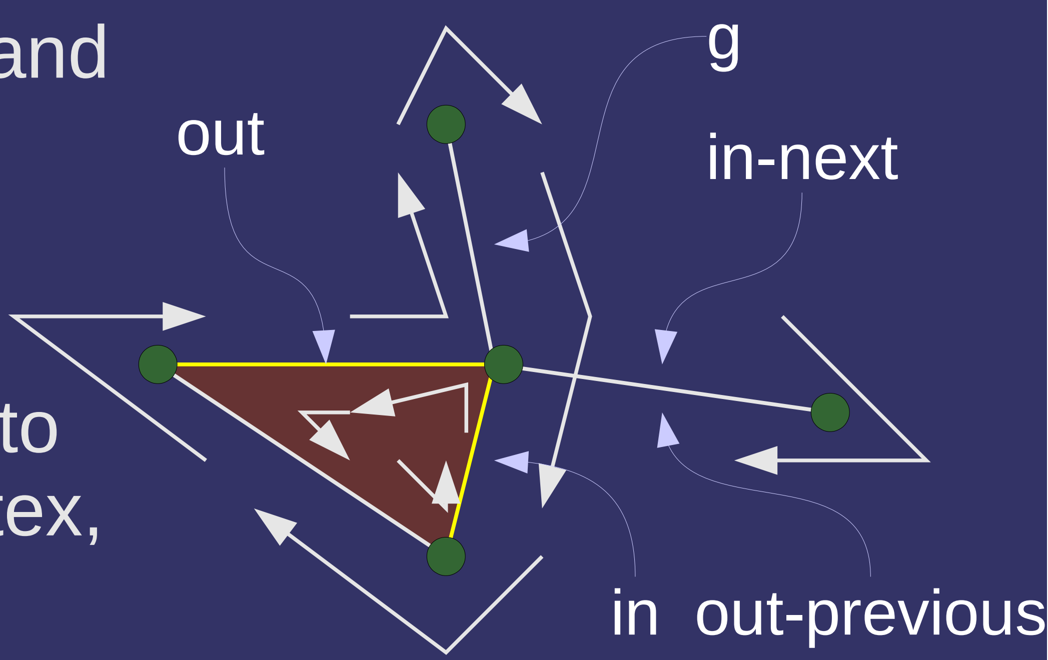
Half-Edge Mesh

- Relink the edges to create the correct relationships
  - Cut the links between *in* and *in-next*, and between *out* and *out-previous*
  - Link *in* and *out*



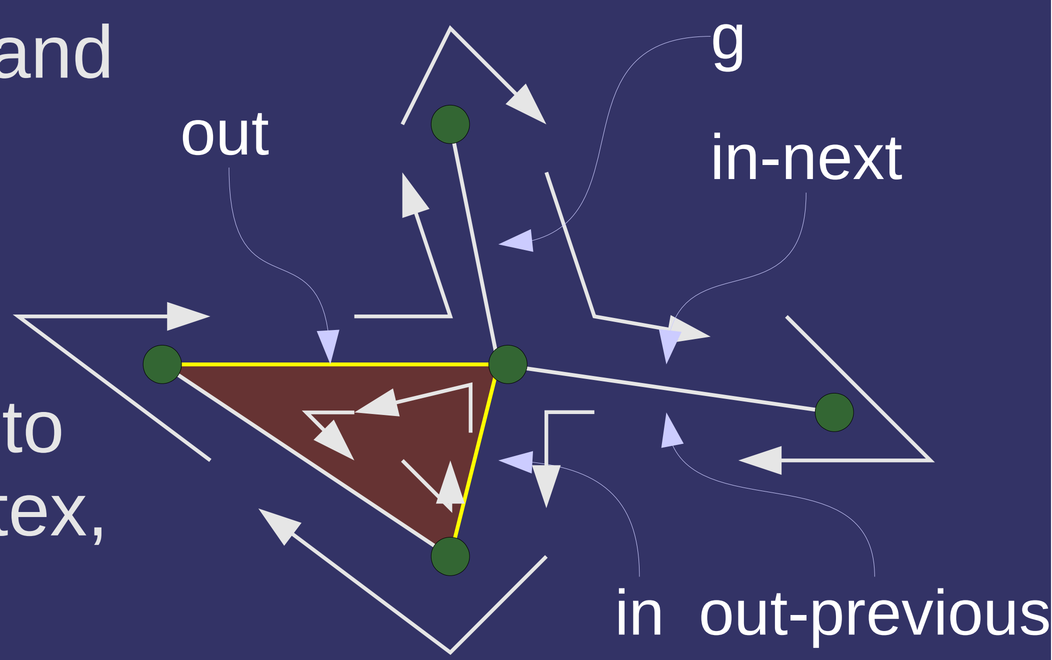
Half-Edge Mesh

- Relink the edges to create the correct relationships
  - Cut the links between *in* and *in-next*, and between *out* and *out-previous*
  - Link *in* and *out*
  - Find a free edge going into *in* and *out's* common vertex, call it *g*
    - This edge must be between *out-sibling* and *in*



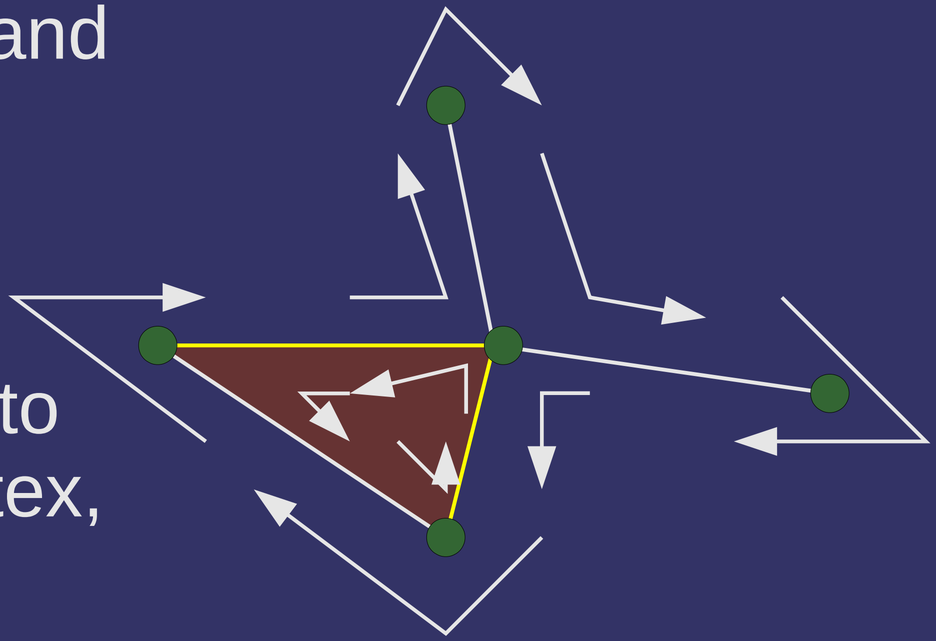
Half-Edge Mesh

- Relink the edges to create the correct relationships
  - Cut the links between *in* and *in-next*, and between *out* and *out-previous*
  - Link *in* and *out*
  - Find a free edge going into *in* and *out's* common vertex, call it *g*
    - This edge must be between *out-sibling* and *in*
  - Link *g* to *in-next*
  - Link *out-previous* to *g-next*



Half-Edge Mesh

- Relink the edges to create the correct relationships
  - Cut the links between *in* and *in-next*, and between *out* and *out-previous*
  - Link *in* and *out*
  - Find a free edge going into *in* and *out's* common vertex, call it *g*
    - This edge must be between *out-sibling* and *in*
  - Link *g* to *in-next*
  - Link *out-previous* to *g-next*



Half-Edge Mesh

- With these primitives, adding a new polygon is easy
  - For all edges, verify that the end point of one edge and the start point of the next edge is the same
  - For all edges, verify that the edge is not already associated with a polygon
  - For all edges, connect the edge to the next edge in the list
  - Allocate a new polygon object and connect all of the edges to it

References

Matt Pharr and Ken Schoemake, ed. *comp.graphics.algorithims FAQ.* Accessed 13 May 2008; available from http://cgafaq.info/wiki/Geometric_data_structures; Internet.

Shadow Volume Geometry

- Once we have a model stored half-edge or winged-edge data structure, how do we generate the shadow volume geometry?

Shadow Volume Geometry

- Once we have a model stored half-edge or winged-edge data structure, how do we generate the shadow volume geometry?
  - For each edge in the mesh:
    - If the either of the edge's polygon pointers is `NULL`, skip the edge
    - Calculate the normal of each polygon sharing the edge, call these n_0 and n_1
    - If n_0 and n_1 are equal, skip the edge
      - This happens if the surfaces are co-planar, and can *never* be on the silhouette
    - Emit a quad of (v_0, n_0), (v_1, n_0), (v_1, n_1), (v_0, n_1)

Fixing Object Geometry

- What about edges with `NULL` polygon pointers?

Fixing Object Geometry

- What about edges with `NULL` polygon pointers?
  - These represent *holes* in the model
    - The Stanford bunny model has several holes in the bottom
  - For each hole, the hole-edges form a ring

Fixing Object Geometry

- What about edges with `NULL` polygon pointers?
  - These represent *holes* in the model
    - The Stanford bunny model has several holes in the bottom
  - For each hole, the hole-edges form a ring
- What can we do with this?

Fixing Object Geometry

- What about edges with `NULL` polygon pointers?
  - These represent *holes* in the model
    - The Stanford bunny model has several holes in the bottom
  - For each hole, the hole-edges form a ring
- What can we do with this?
  - Walk the hole-edge ring and insert *new* edges between each pair of hole-edges
  - Each new edge will form a triangle that fills part of the hole
  - Do this step *before* generating shadow volume geometry

Next week...

- Advanced shadow volume techniques:
  - Fixing z-pass and z-fail with ZP+
  - Soft shadows using shadow volumes
  - Hardware based optimizations:
    - Depth clamping
    - Depth bounds testing

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.