VGP351 – Week 3

- Agenda:
  - Quiz #1
  - Hidden surface removal / occlusion
    - Backface culling
    - Painters algorithm
    - Z-buffer
    - Frustum culling

Hidden Surface Removal

- Why?

Hidden Surface Removal

- Why?
  - Correctness: if object A is behind object B, object A should not obscure object B
  - Performance: don't spend time drawing things that cannot be seen
    - Obscured objects
    - Polygons on the “backside” of objects
    - Objects outside the camera's view

Backface Culling

- The faces on the back side of this cube can't be seen because they face *away* from the viewer
  - There are two common ways to determine that polygon faces away from viewer

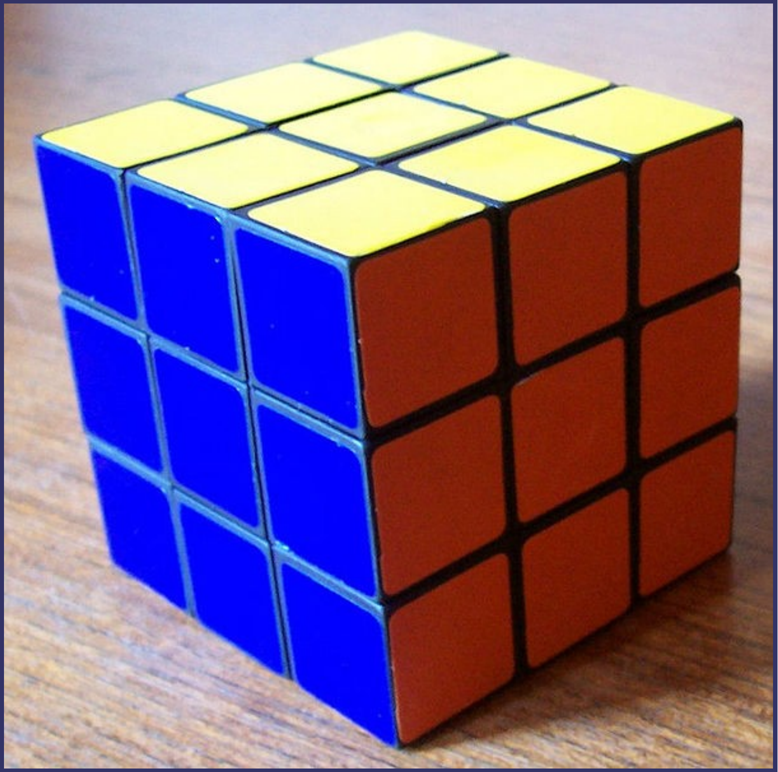


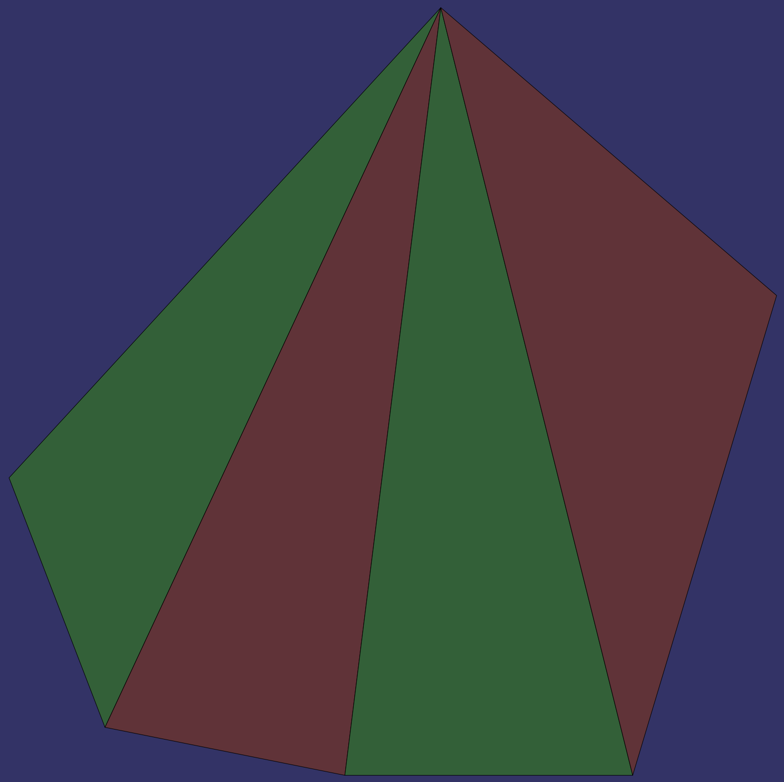
Image from http://en.wikipedia.org/wiki/File:Cubo_rubik_2.jpg

Backface Culling

- Compare the direction of the surface normal with the viewing direction
  - If $\mathbf{n} \cdot \mathbf{v} > 0$, the surface faces away from the camera
- Several problems with this method:
  - Requires that you have *surface* normals
  - Must be implemented differently for different types of viewing projections

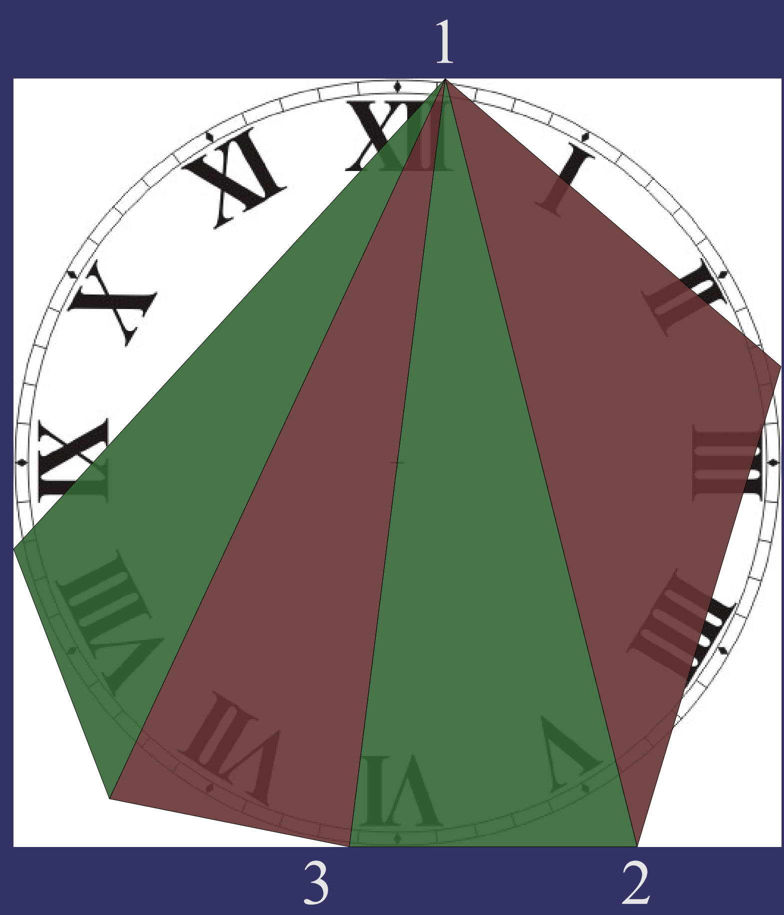
Backface Culling

- After projection to 2D, it is possible to determine if vertices are ordered clockwise or counter-clockwise



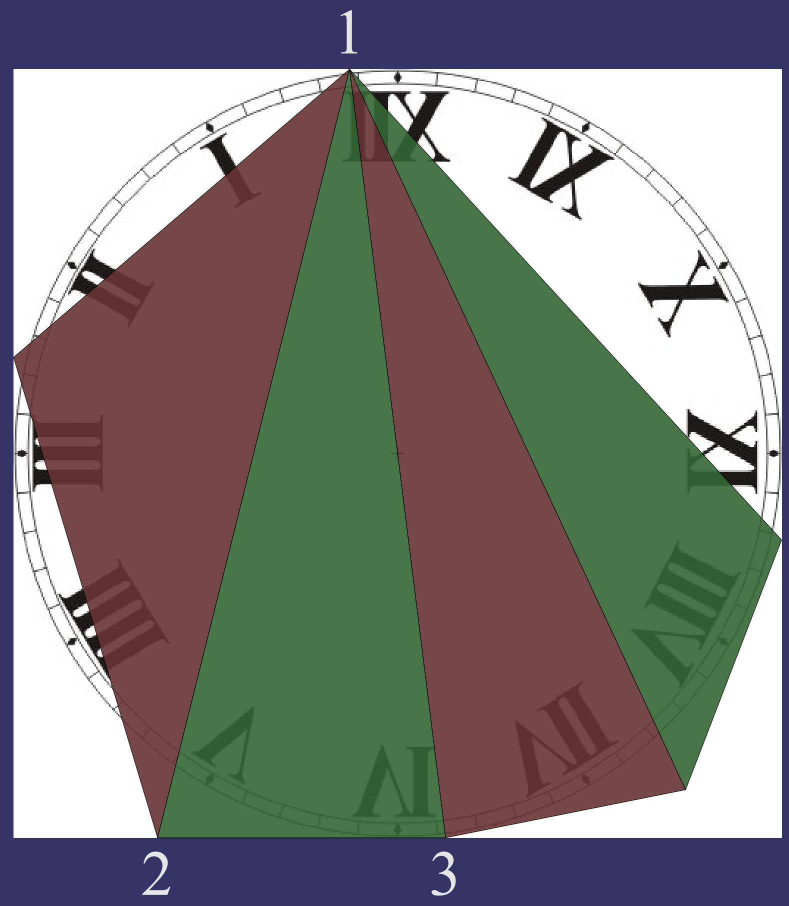
Backface Culling

- After projection to 2D, it is possible to determine if vertices are ordered clockwise or counter-clockwise



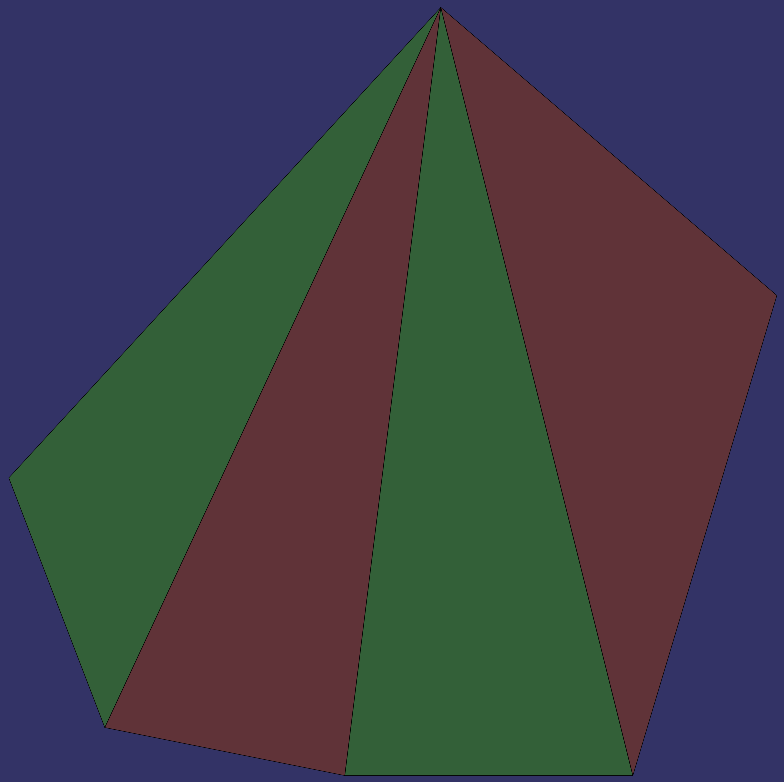
Backface Culling

- After projection to 2D, it is possible to determine if vertices are ordered clockwise or counter-clockwise



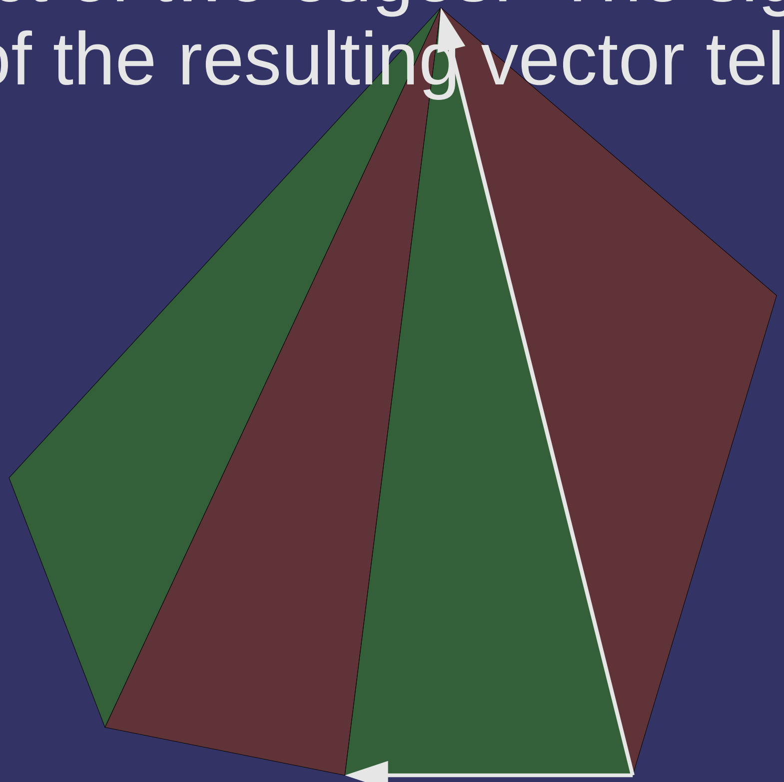
Backface Culling

- After projection to 2D, it is possible to determine if vertices are ordered clockwise or counter-clockwise
  - How?



Backface Culling

- After projection to 2D, it is possible to determine if vertices are ordered clockwise or counter-clockwise
  - Cross-product of two edges! The sign of the Z-component of the resulting vector tells you the facing



Backface Culling

- Backface culling is enabled with:

```
glEnable(GL_CULL_FACE);
```

- Frontface orientation is selected with:

```
glFrontFace(GL_CW);
```

  - Clockwise ordered polygons are considered front-facing

```
glFrontFace(GL_CCW);
```

  - Counter-clockwise ordered polygons are considered front-facing
  - This is the default setting

Depth Ordering

- Just drawing objects in arbitrary order gives incorrect results

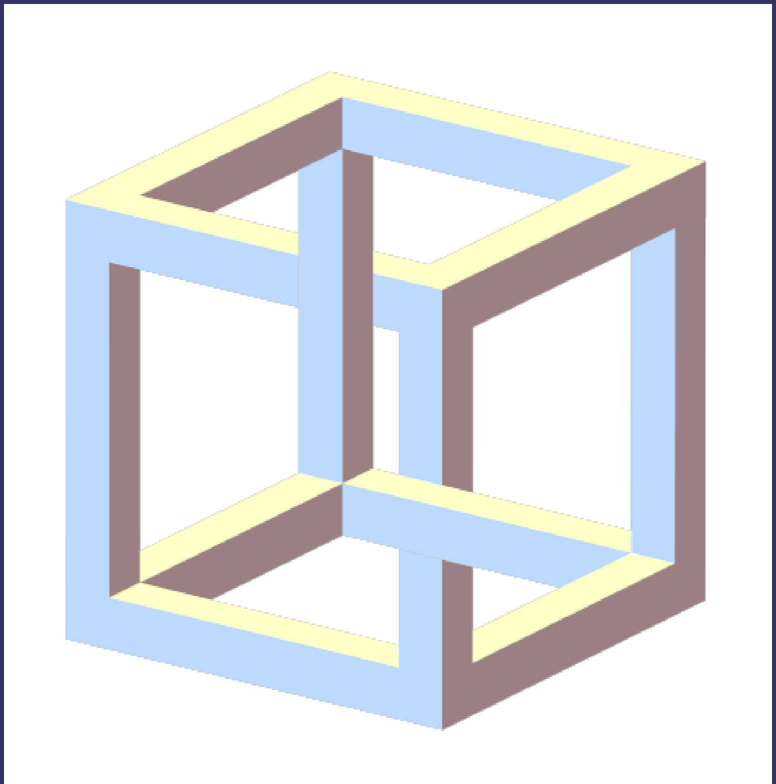


Image from http://www.planetperplex.com/en/item253

Depth Ordering

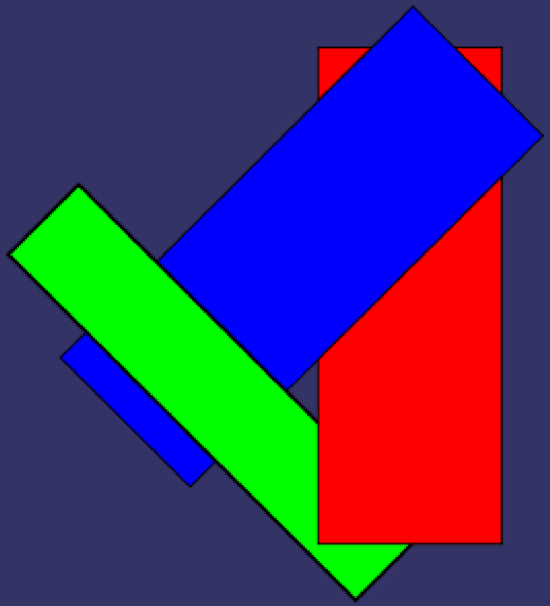
- Just drawing objects in arbitrary order gives incorrect results
- Several geometric solutions exist
  - Painter's algorithm
  - BSP tree
  - Warnock's algorithm
    - We won't actually talk about this algorithm
  - Ray tracing
    - We'll talk about this later in the term

Painter's Algorithm

- Algorithm traditionally used *before* 3D accelerators:

  The name "painter's algorithm" refers to the technique employed by many painters of painting distant parts of a scene before parts which are nearer....The [algorithm] sorts all the polygons in a scene by their depth and then paints them in this order, furthest to closest.[1]

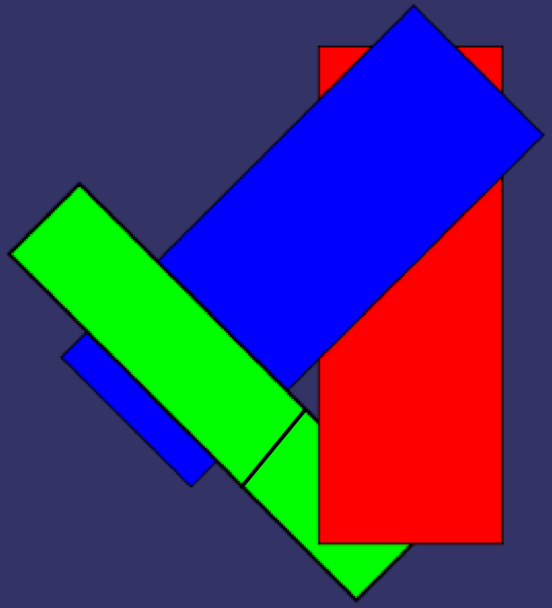
- Suffered from many problems:
  - The sorting step is slow
  - How to deal with mutually overlapping polygons?



1 http://en.wikipedia.org/wiki/Painter%27s_algorithm

BSP Tree

- Binary tree where each node splits space
  - Each node contains an n-dimensional split-plane
  - One child is in the positive-space of the plane and the other is in the negative-space
  - If a polygon is added to a node crosses the split-plane, partition the polygon at the plane



- Resulting tree can be traversed *in order* quickly
  - This is (part of) the method that Quake and Quake II use for hidden surface removal

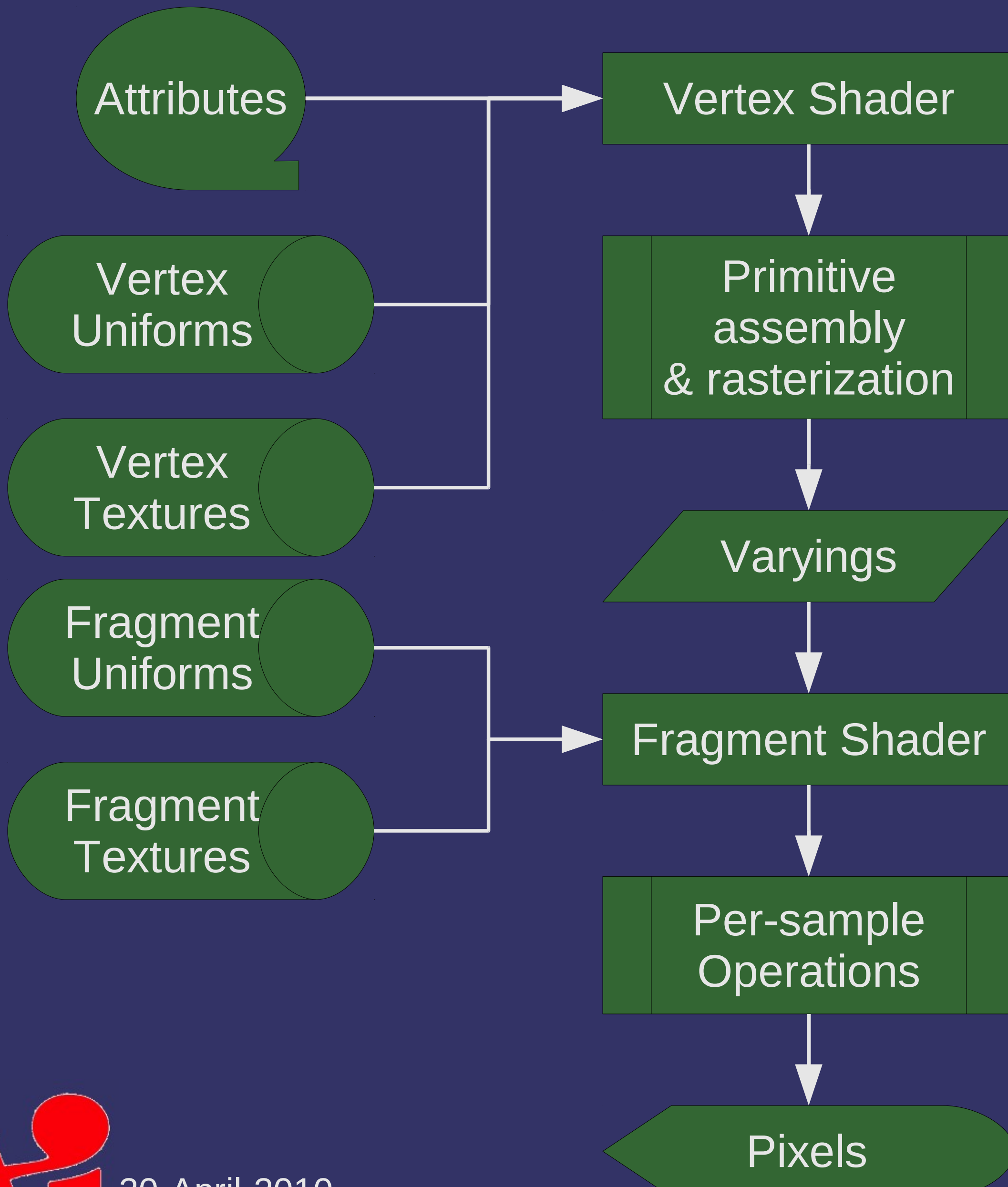
BSP Tree

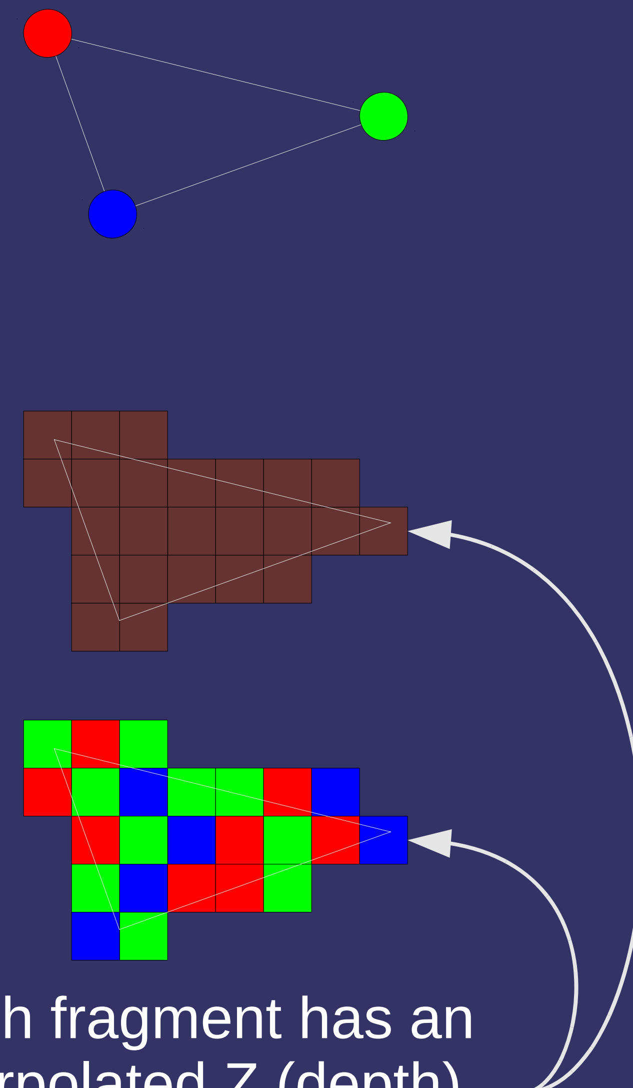
- Even though traversal is fast, there are several drawbacks:
  - Splitting polygons can create lots of extra data
  - Splitting polygons can create cracks due to numeric round-off
  - Creating good trees is *very* expensive!
    - Largely useless for scenes with lots of dynamic objects
    - This is why you can't destroy walls in most 3D games. :)

Depth Ordering

- Geometric solutions to the visibility problem have largely proven ineffective
  - The usual solution is an image-space solution: the depth buffer

Pipeline Data Flow

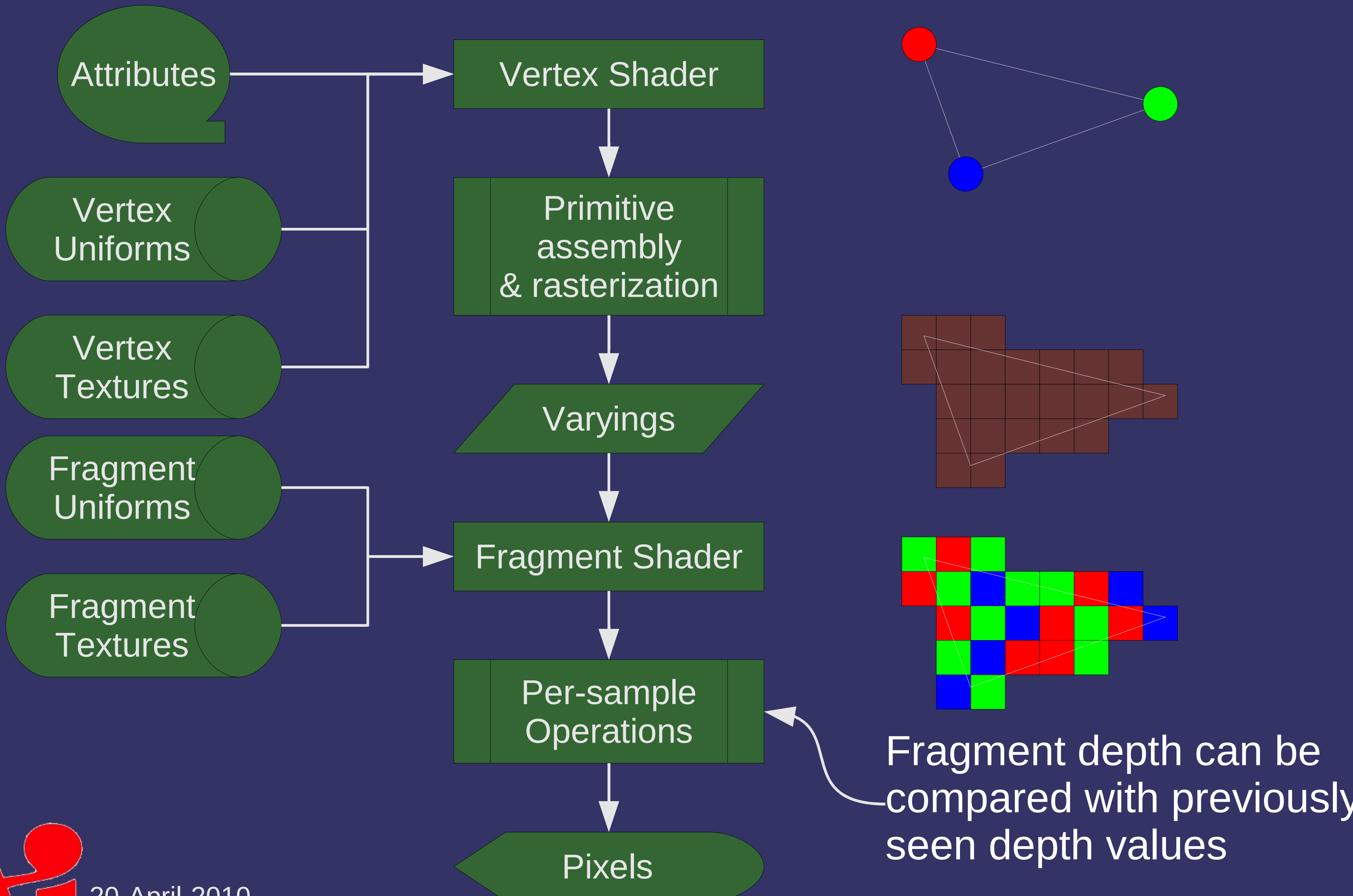




Each fragment has an interpolated Z (depth) value

20-April-2010

Pipeline Data Flow



Depth Buffer

- Depth buffering isn't perfect
  - Differences in interpolation values can lead to errors...

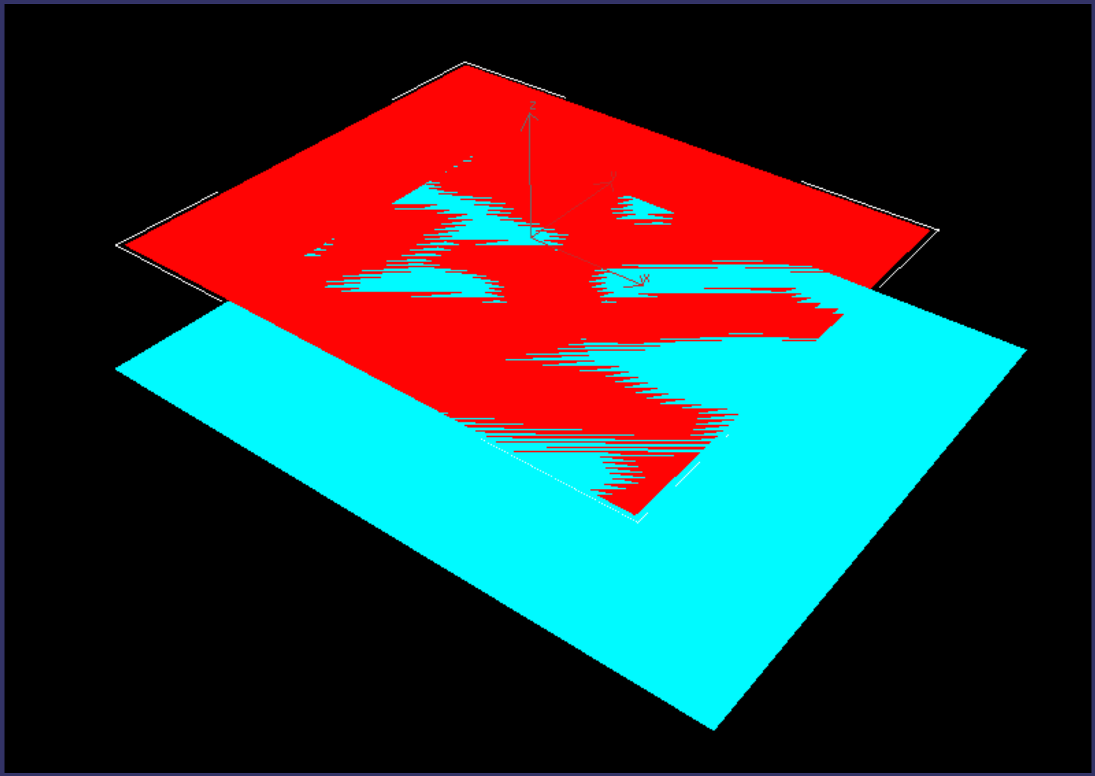


Image from http://en.wikipedia.org/wiki/File:Z-fighting.png

Depth Buffer in OpenGL

- Depth test compares the depth value of each fragment of a polygon with the depth value stored at each pixel
  - If the test passes, the fragment is drawn
  - If the test fails, the fragment is discarded
- To use a depth buffer, we have to allocate one:

```
SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 24);
```

Common maximum depth buffer size

Depth Buffer in OpenGL

- Depth test has an enable:

  `glEnable(GL_DEPTH_TEST);`

- Must also set the comparison mode:

  `glDepthFunc(GLenum mode);`

  – `mode` is one of `GL_LESS`, `GL_LEQUAL`, `GL_GREATER`, `GL_GEQUAL`, `GL_EQUAL`, `GL_NOTEQUAL`, `GL_NEVER`, `GL_ALWAYS`

Depth Buffer in OpenGL

- Clear the depth buffer just like the color buffer:

```
glClear(GL_COLOR_BUFFER_BIT |
        GL_DEPTH_BUFFER_BIT);
```

- Set the clear value:

```
void glClearDepth(GLclampd depth);
```

Special type! Means that a floating-point value from 0.0 to 1.0 is required.

Perspective Projection

$$\mathbf{M}_p = \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \times far \times near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

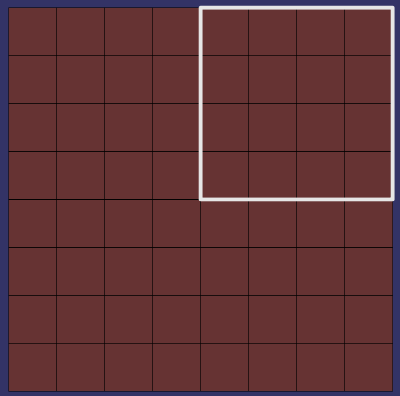
This row remaps Z values on the range [-*near*, -*far*] to [-1, 1].

Depth Buffer Acceleration

- Per-pixel depth comparison in complex environments is *very* expensive
- Many common optimizations exist:
  - Test depth before the fragment shader
    - Saves cost of running fragment shader on occluded fragments
    - Called “early Z”
    - Cannot be used if the fragment shader modifies the depth value
  - Hierarchical depth buffer
  - Depth buffer compression
  - Fast Z clear

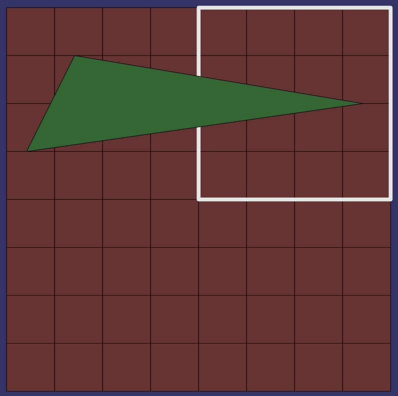
Hierarchical Depth Buffer

- Depth buffer is stored by tiles
  - Store the minimum (or maximum) value of each tile



Hierarchical Depth Buffer

- Depth buffer is stored by tiles
  - Store the minimum (or maximum) value of each tile
- Compare an entire polygon against the tiles that it overlaps
  - Allows rejection of entire polygons or large portions of a polygon very quickly



Depth Buffer Compression

- Several observations:
  - Most of the depth buffer will contain the clear value
  - Most depth values in a block will be close to the near value in the hierarchical buffer
  - Most depth values in a block will be close to the other values in the block
- Individual blocks can be stored more compactly
  - Most methods store one full precision value and lower precision per-pixel deltas from that value

Fast Z Clear

- Writing the same value to all locations in the depth buffer takes a lot of bandwidth
  - Store a single bit per $n \times n$ block
  - Set that single bit per block when `glClear` is called
    - For this to work, clear all the buffers with a single call to `glClear`
  - When rendering, if the bit is set, use the clear value for the whole block
- Why does this work?
  - The block size matches the cache line size
  - Data is written back one cache line at a time, so writing the cleared block back adds no extra cost

View-volume Culling

- Determine that an object is entirely outside the viewing volume
  - Usually an approximation called a *bounding volume* is used to represent the object
  - This early culling allows us to avoid even sending the object to the graphics library

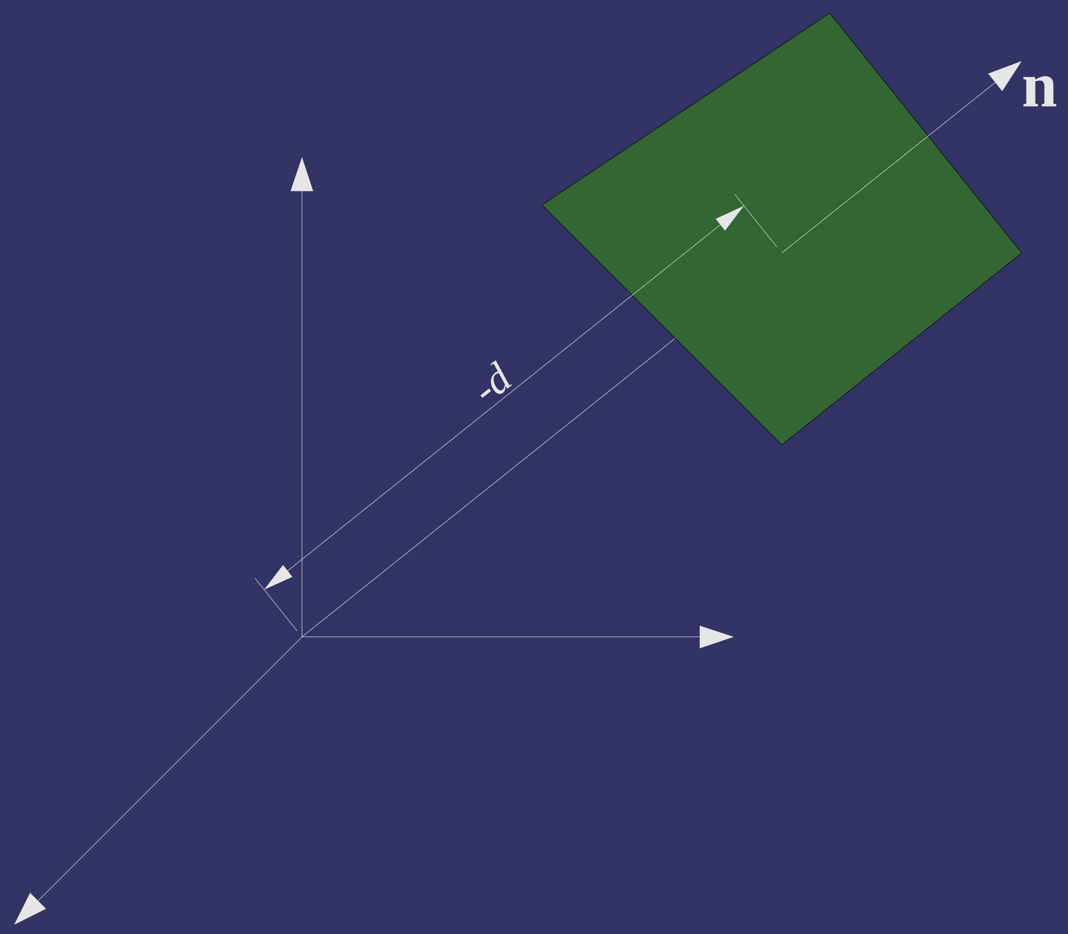
Plane Equation

- Arbitrary planes in a space are represented by a *plane equation* with the following form:

$$(\mathbf{n}_p \cdot \mathbf{p}) + d_P = 0$$

  - $\mathbf{n}_p$ is the normal of the plane
  - $-d_p$ is the distance from the origin to the plane in the direction of the normal

Plane Equation



20-April-2010

Plane Equation

- If we know three non-colinear points on the plane, the plane equation is easy to calculate
  - Calculate the normal from the cross-product of two edge vectors:

$$\hat{\mathbf{v}}_0 = \mathbf{v}_0 - \mathbf{v}_1$$

$$\hat{\mathbf{v}}_1 = \mathbf{v}_2 - \mathbf{v}_1$$

$$\mathbf{n}_\mathrm{p} = \frac{\hat{\mathbf{v}}_0 \times \hat{\mathbf{v}}_1}{|\hat{\mathbf{v}}_0 \times \hat{\mathbf{v}}_1|}$$

  - Calculate d using the dot product:

$$-d = \mathbf{n}_\mathrm{p} \cdot \mathbf{v}$$

    - $\mathbf{v}$ is *any* point on the plane

Plane Equation

- Using the equation of a plane, we can determine which “side” of the plane a point is on

$$(\mathbf{n}_p \cdot \mathbf{p}) + d = k$$

  - $\mathbf{p}$ is a point to be tested
  - If $k = 0$, then $\mathbf{p}$ is on the plane
  - If $k < 0$, then $\mathbf{p}$ is “inside” the plane
    - Technically, it is in the negative half-space
  - If $k > 0$, then $\mathbf{p}$ is “outside” the plane
    - Technically, it is in the positive half-space

View-volume Culling

- Observation: a view-volume is made from 6 planes
  - If a point is in the positive half-space of *any* of the 6 planes, it is outside the view volume
- If we have a bounding sphere for each object in the scene, we can use the point-in-volume test
  - For each object, “grow” the frustum by the radius of the sphere
  - Test the center of the sphere against the new planes

$$\left(\mathbf{n}_p \cdot \mathbf{c}\right) + \left(d - r\right) = k$$

Further Reading

Ulf Assarsson and Tomas Möller, "Optimized View Frustum Culling Algorithms for Bounding Boxes," *journal of graphics tools*, 5(1), pp 9-22, 2000. http://www.cse.chalmers.se/~uffe/vfc_bbox.pdf

http://www.realtimerendering.com/intersections.html

Next week...

- Lighting!
  - Lighting models
  - Shading methods
  - Types of lights
- Assignments:
  - Start assignment #2, part 1

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.

Quake and Quake II are trademarks of id Software.

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