Alpha Blending and Introduction to Geometry
• Alpha Blending

• The Graphics Pipeline Revisited

• Introduction to Geometry
Another common image format: RGBA
- Alpha channel specifies ‘opacity’ of object
- Basically how transparent it is
- Most common encoding is 8-bits per channel (0-255)

Compositing A over B != B over A
- Consider the extreme case of two opaque objects...

\[ \alpha = 1 \]
\[ \alpha = 3/4 \]
\[ \alpha = 1/2 \]
\[ \alpha = 1/4 \]
\[ \alpha = 0 \]

fully opaque
fully transparent

[ koala over nyc ]
[ nyc over...koala? ]

where is the koala...
Non-Premultiplied Alpha

- **Goal**: Composite image $B$ with alpha $\alpha_B$ over image $A$ with alpha $\alpha_A$
  
  $A = (A_r, A_g, A_b)$
  $B = (B_r, B_g, B_b)$

- **Composite RGB**: what $B$ lets through
  
  $C = \alpha_B B + (1 - \alpha_B)\alpha_A A$

- **Composite Alpha**:
  
  $\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$

Two different equations is inefficient!!
Premultiplied Alpha

• **Goal:** Composite image $B$ with alpha $\alpha_B$ over image $A$ with alpha $\alpha_A$

\[
A' = (\alpha_A A_r, \alpha_A A_g, \alpha_A A_b, \alpha_A) \\
B' = (\alpha_B B_r, \alpha_B B_g, \alpha_B B_b, \alpha_B)
\]

• **Composite RGBA:**

\[
C' = B' + (1 - \alpha_B) A'
\]

• **Un-Premultiply for Final Color:**

\[
(C_r, C_g, C_b, \alpha_C) \Rightarrow (C_r/\alpha_C, C_g/\alpha_C, C_b/\alpha_C)
\]
Why Premultiplied Matters [Upsample]

New background \( A (\alpha_A = 1) \)  

\( B \) over \( A \)  

\( B \) over \( A \) (premultiplied)

Known as fringing

Something isn’t right...
Why Premultiplied Matters [Downsample]

original
regular
premultiplied
downsampled
composite

[ RGB ]

[ A ]
Closed Under Composition

• **Goal:** Composite bright red image \( B \) with alpha 0.5 over bright red image \( A \) with alpha 0.5

\[
A = (1, 0, 0, 0.5) \\
B = (1, 0, 0, 0.5)
\]

• **Non-Premultiplied:**

\[
0.5 \times (1,0,0) + (1 - 0.5) \times 0.5 \times (1,0,0) \\
= (0.75, 0, 0)
\]

• **Premultiplied:**

\[
0.5 \times (0.5,0,0,0.5) + (1 - 0.5) \times (0.5,0,0,0.5) \\
= (0.75, 0, 0, 0.75) \\
\text{divide out alpha} \\
= (1, 0, 0)
\]
Blend Methods

When writing to color buffer, can use any blend method

\[
\begin{align*}
D_{\text{RGBA}} &= S_{\text{RGBA}} + D_{\text{RGBA}} \\
D_{\text{RGBA}} &= S_{\text{RGBA}} - D_{\text{RGBA}} \\
D_{\text{RGBA}} &= -S_{\text{RGBA}} + D_{\text{RGBA}} \\
D_{\text{RGBA}} &= \min(S_{\text{RGBA}}, D_{\text{RGBA}}) \\
D_{\text{RGBA}} &= \max(S_{\text{RGBA}}, D_{\text{RGBA}}) \\
D_{\text{RGBA}} &= S_{\text{RGBA}} + D_{\text{RGBA}} \times (1 - S_{\text{A}})
\end{align*}
\]

Blend Add
Blend Subtract
Blend Reverse Subtract
Blend Min
Blend Max
Blend Over

\( S_{\text{RGBA}} \) and \( D_{\text{RGBA}} \) are pre-multiplied
Updated Depth Buffer (Z-buffer) Sample Code

draw_sample(x, y, d, c) // new depth d & color c at (x,y)
{
  if (d < zbuffer[x][y]) {
    // triangle is closest object seen so far at this sample point. Update depth and color buffers.
    zbuffer[x][y] = d;
    color[x][y] = c.rgba + (1-c.a) * color[x][y];
  }
  // otherwise, we've seen something closer already;
  // don't update color or depth
}

Assumes color[x][y] and c are both premultiplied.

Triangles must be rendered back to front!
A over B != B over A
• For mixtures of opaque and transparent triangles:
  
  • **Step 1:** render opaque primitives (in any order) using depth-buffered occlusion
    • If pass depth test, triangle overwrites value in color buffer at sample
    • Depth **READ** and **WRITE**
  
  • **Step 2:** disable depth buffer update, render semi-transparent surfaces in back-to-front order.
    • If pass depth test, triangle is composited **OVER** contents of color buffer at sample
    • Depth **READ** only
• Alpha Blending

• The Graphics Pipeline Revisited

• Introduction to Geometry
Now Let’s Put It All Together!

The “Simpler” Graphics Pipeline

Transform/position objects in the world → Project objects onto the screen → Sample triangle coverage

→ Combine samples into final image (depth, alpha, …)

→ Sample texture maps / evaluate shaders

→ Interpolate triangle attributes at covered samples
The Inputs

positions = {
    v0x, v0y, v0z,
    v1x, v1y, v1x,
    v2x, v2y, v2z,
    v3x, v3y, v3x,
    v4x, v4y, v4z,
    v5x, v5y, v5x
};

texcoords = {
    v0u, v0v,
    v1u, v1v,
    v2u, v2v,
    v3u, v3v,
    v4u, v4v,
    v5u, v5v
};

Object-to-camera-space transform $T \in \mathbb{R}^{4 \times 4}$
Perspective projection transform $P \in \mathbb{R}^{4 \times 4}$
Output image $(W, H)$
Step 1: Transform

Transform triangle vertices into camera space
Step 2: Perspective Projection

Apply perspective projection transform to transform triangle vertices into normalized coordinate space.
Step 3: Clipping

Discard triangles completely outside cube.
Clip triangles partially in cube.
Step 4: Transform To Screen Coordinates

Perform homogeneous divide. Transform vertex xy positions from normalized coordinates into screen coordinates (based on screen \([w, h]\)).

\((0, 0)\) \hspace{2cm} \((w, h)\)
Step 5: Sample Coverage

Check if samples lie inside triangle.
Evaluate depth and barycentric coordinates at all passing samples.
Step 6: Compute Color

Texture lookups, color interpolation, etc.

\[ [u(x,y), v(x,y)] \]
Step 7: Depth Test

Check depth and update depth if closer primitive found.
(can be disabled)
Step 8: Color Blending

Update color buffer with correct blending operation.
The “Real” Graphics Pipeline

Doesn’t look much different than what we discussed...
• Alpha Blending

• The Graphics Pipeline Revisited

• Introduction to Geometry
• Alpha Blending

• The Graphics Pipeline Revisited

• Introduction to Geometry
  • Implicit & Explicit Geometry
  • Manifold Geometry
  • Local Geometric Operations
Some Motivation

“I hate meshes. I cannot believe how hard this is. Geometry is hard.”

“why won’t you subdivide”

-- David Baraff
Senior Research Scientist
Pixar Animation Studios
(also a former CMU prof.)
What Is Geometry?

**ge-o-m-e-t-r-y** /jēˈæmətrē/ *n.*
1. The study of shapes, sizes, patterns, and positions.
2. The study of spaces where some quantity (lengths, angles, etc.) can be *measured*.

Remember that Computer Graphics is just operating on a bunch of numbers. If we can measure it, we can represent it as numbers on our computer!
How To Represent Geometry

[ IMPLICIT ]
\[ x^2 + y^2 = 1 \]

[ LINGUISTIC ]
“unit circle”

[ EXPLICIT ]
\( (\cos \theta, \sin \theta) \)

which is best?

[ TOMOGRAPHIC ]
(constant density)

[ CURVATURE ]
\( \kappa = 1 \)

[ SYMMETRIC ]

[ DYNAMIC ]
\[ \frac{d^2}{dt^2} x = -x \]

[ DISCRETE ]
\( n \to \infty \)
How To Represent Humans
How To Represent Water
How To Represent Cloth
How To Represent Machines
How To Represent This Thing
Many Ways To Encode Geometry

• Explicit:
  • point cloud
  • polygon meshes
  • subdivision surfaces
  • NURBS

• Implicit:
  • level set
  • constructive solid geometry
  • algebraic surface
  • L-systems
  • Fractals

• Not one best geometric representation!
  • Each is suited for a different task
  • Tradeoffs between:
    • Accuracy
    • Memory
    • Performance (searching/operating)
Implicit Geometry

- Points aren’t known directly, but satisfy some relationship
  - Example: unit sphere is all points such that \( x^2 + y^2 + z^2 = 1 \)
- More generally, in the form \( f(x,y,z) = 0 \)
- Finding example points is **hard**
  - Requires solving equation
- Checking if points are inside/outside is **easy**
  - Just evaluate the function with a given point
Explicit Geometry

- All points are given directly

- More generally:
  \[ f : \mathbb{R}^2 \rightarrow \mathbb{R}^3; (u, v) \mapsto (x, y, z) \]
  - Given any \((u, v)\), we can find a point on the surface
  - Can limit \((u, v)\) to some range
    - Example: triangle with barycentric coordinates

- Finding example points is \textbf{easy}
  - We are given them for free

- Checking if points are inside/outside is \textbf{hard}
  - We are given the output values and need to find input values that satisfy the geometry
What does *easy* and *hard* mean?
Implicit Geometry [Hard]

• Given the unit sphere:

\[ f(x, y, z) = x^2 + y^2 + z^2 = 1 \]

• Find a point that exists on it.

• **Answer:** (1,0,0)
  • Not so difficult, but how did you arrive at the answer?
  • We are given a constraint, and need to find parameters \((x, y, z)\) that satisfy the constraint
    • Keep guessing and checking
Implicit Geometry [Easy]

• Given the unit sphere:

\[ f(x, y, z) = x^2 + y^2 + z^2 = 1 \]

• Find if the point (0.75, 0.5, 0.25) lives inside it.

• **Answer:** yes!
  • \[ f(0.75, 0.5, 0.25) = 0.75^2 + 0.5^2 + 0.25^2 = 0.875 < 1 \]
  • Easy to check! Just evaluate the sign of the function at the desired point
Explicit Geometry [Easy]

- Given the torus:
  
  \[ f(u, v) = ((2 + \cos u) \cos v, (2 + \cos u) \sin v, \sin u) \]

- Find a point that exists on it.

- **Answer**: (3,0,0)
  - Just plug in any value of \((u, v)\)!
  - We plugged in \((u, v) = (0,0)\)
Explicit Geometry [Hard]

• Given the torus:

\[ f(u, v) = ((2 + \cos u) \cos v, (2 + \cos u) \sin v, \sin u) \]

• Find if the point (1.96, -0.39, 0.9) lives inside it.

• **Answer:** no, I’m not computing that
  • We are given a constraint, and need to find parameters \((u, v)\) that satisfy the constraint
    • Keep guessing and checking
Let’s look at some implicit examples...
Algebraic Surfaces [Implicit]

- A surface built with algebra
  - Generally thought of as a surface where points are some radius $r$ away from another point/line/surface
- [+] Generates smooth/symmetric surfaces
- [-] Cannot generate impurities/deformations

![Equations](https://via.placeholder.com/150)

$$x^2 + y^2 + z^2 = 1$$

$$\left( R - \sqrt{x^2 + y^2} \right)^2 + z^2 = r^2$$

$$\left( x^2 + \frac{9y^2}{4} + z^2 - 1 \right)^3 = \frac{x^2z^3 + \frac{9y^2z^3}{80}}$$
Constructive Solid Geometry [Implicit]

- Build more complicated shapes via Boolean operations
  - Basic operations:

- Can be used to form complex shapes!
Blobby Surfaces [Implicit]

- Instead of Booleans, gradually blend surfaces together:

  ![Blobby Surfaces Diagram]

- Easier to understand in 2D:

  \[
  \phi_p(x) := e^{-|x-p|^2} \quad \text{(Gaussian centered at } p) \\
  f := \phi_p + \phi_q \quad \text{(Sum of Gaussians centered at different points)}
  \]
Level Set Methods [Implicit]

- Store a grid of values approximating function

<table>
<thead>
<tr>
<th>-.55</th>
<th>-.45</th>
<th>-.35</th>
<th>-.30</th>
<th>-.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>-.30</td>
<td>-.25</td>
<td>-.20</td>
<td>-.10</td>
<td>-.10</td>
</tr>
<tr>
<td>-.20</td>
<td>-.15</td>
<td>-.10</td>
<td>.10</td>
<td>.15</td>
</tr>
<tr>
<td>-.05</td>
<td>.10</td>
<td>.05</td>
<td>.25</td>
<td>.35</td>
</tr>
<tr>
<td>.15</td>
<td>.20</td>
<td>.25</td>
<td>.55</td>
<td>.60</td>
</tr>
</tbody>
</table>

\[ f(x) = 0 \]

- Surface is found where interpolated values equal zero
- [+] Provides much more explicit control over shape
- [-] Runs into problems of aliasing!
Fractals [Implicit]

- No precise definition; exhibit self-similarity, detail at all scales
- [+] New “language” for describing natural phenomena
- [-] Hard to control shape!
Let’s look at some explicit examples...
Point Cloud [Explicit]

- A list of points \((x, y, z)\)
  - Often augmented with normals

- [+] Easily represent any kind of geometry
- [+] Easy to draw dense cloud (>>1 point/pixel)
- [+] Easy for simulation
- [-] Large lookup time
- [-] Large memory overhead
  - Hard to interpolate undersampled regions
  - Hard to do processing / simulation /
  - Result is just as good as the scan
Triangle Mesh [Explicit]

- [+] Easy interpolation with good approximation
  - Use barycentric interpolation to define points inside triangles

- [-] Large memory overhead
  - Store vertices as triples of coordinates \((x,y,z)\)
  - Store triangles as triples of indices \((i,j,k)\)

- Polygonal Mesh: shapes do not need to be triangles
  - Ex: quads

\[
\begin{array}{ccc}
\text{VERTICES} & \text{TRIANGLES} \\
\begin{bmatrix} x & y & z \\ 0: & -1 & -1 & -1 \\ 1: & 1 & -1 & 1 \\ 2: & 1 & 1 & -1 \\ 3: & -1 & 1 & 1 \end{bmatrix} & \begin{bmatrix} i & j & k \\ 0 & 2 & 1 \\ 0 & 3 & 2 \\ 3 & 0 & 1 \\ 3 & 1 & 2 \end{bmatrix} \\
\end{array}
\]

\[
p = \phi_ip_i + \phi_jp_j + \phi_kp_k
\]

\[
\phi_i + \phi_j + \phi_k = 1, \quad \phi_i, \phi_j, \phi_k > 0
\]
- Alpha Blending

- The Graphics Pipeline Revisited

- Introduction to Geometry
  - Implicit & Explicit Geometry
  - Manifold Geometry
  - Local Geometric Operations
Manifold Assumption

- A mesh is manifold if and only if it can exist in real life
  - Important for simulation/3D printing

- Everything in real life has volume to it
  - Likewise, every manifold surface has some volume it encases
  - Allows us to think of manifold surfaces as ‘shells’ to an inner volume
    - **Example:** M&Ms

- Everything in real life, when zoomed in far enough, should be able to have a rectangular coordinate grid
  - Likewise, every manifold surface should be planar when zoomed in far enough
    - **Example:** Planet Earth
Manifold Properties

• For polygonal surfaces, check for “fins” and “fans”

• Every edge is contained in only two polygons (no “fins”)
  • The extra 3\textsuperscript{rd} or 4\textsuperscript{th} or 5\textsuperscript{th} or so forth polygon is the fin of a fish

• The polygons containing each vertex make a single “fan”
  • We should be able to loop around the faces around a vertex in a clear way
Manifold Check
Manifold Check

**https://github.com/rlguy/Blender-FLIP-Fluids/wiki/Manifold-Meshes**
Planes Are Not Manifold

• Each edge of a plane only touches 1 polygon
  • Breaks the “fin” constraint

• More intuitively: **no notion of thickness!**
  • Can not be represented in real life
  • Paper (best approximation of plane) still has thickness

• **How to make manifold:** add a second polygon that overlaps with the first plane, connecting all the edges
  • Messy, two polygon will overlap, but will fix the manifold issue

• **How to make manifold:** add a new type of edge denoting it as a boundary
  • The “boundary” edge
Boundary Edges

- Objects in real life (Ex: pants) have boundaries
  - Boundary geometry loops around to create the inner seams of the pants
  - The volume enclosed by pants are not where your legs go, but the physical thickness of the pants

- Representing both the inside and outside of pants is expensive!
  - Use boundary edges

- A boundary edge has 1 polygon per edge
  - This does not mean planes are manifold! This just gives us a way to represent complex manifold geometry as simpler non-manifold geometry
What are some ways to describe the connectivity of geometry?
Polygon Soup

- Most basic idea imaginable:
  - For each triangle, just store three coordinates
  - No other information about connectivity
  - Not much different from point cloud
    - A “Triangle cloud”?

- Pros:
  - [+] Really stupid simple

- Cons:
  - [-] Really stupid
  - [-] Redundant storage of vertices
  - [-] Very difficult to find neighboring polygons

(x0, y0, z0)  x1, y1, z1  x3, y3, z3
(x1, y1, z1)  x2, y2, z2  x3, y3, z3
(x2, y2, z2)  (x0, y0, z0)
(x3, y3, z3)

(x0, y0, z0)  x1, y1, z1  x3, y3, z3
x1, y1, z1  x2, y2, z2  x3, y3, z3
Adjacency List

• A little more complicated:
  • Store triples of coordinates \((x,y,z)\)
  • Store tuples of indices referencing the coordinates needed to build each triangle

• Pros:
  • [+] No duplicate coordinates
  • [+] Lower memory footprint
  • [+] Easy to keep geometry manifold
  • [+] Supports nonmanifold geometry
  • [+] Easy to change connectivity of geometry

• Cons:
  • [-] Very difficult to find neighboring polygons
  • [-] Difficult to add/remove mesh elements

<table>
<thead>
<tr>
<th>VERTICES</th>
<th>POLYGONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x) (y) (z)</td>
<td>(i) (j) (k)</td>
</tr>
<tr>
<td>-1 -1 -1</td>
<td>0 2 1</td>
</tr>
<tr>
<td>1 -1 1</td>
<td>0 3 2</td>
</tr>
<tr>
<td>1 1 -1</td>
<td>3 0 1</td>
</tr>
<tr>
<td>-1 1 1</td>
<td>3 1 2</td>
</tr>
</tbody>
</table>
Incidence Matrices

- If we want to know our neighbors, let’s store them:
  - Store triples of coordinates \((x,y,z)\) Store incidence matrix between vertices + edges, and edges + faces
    - 1 means touch, 0 means no touch
    - Store as sparse matrix

- Pros:
  - [+ ] No duplicate coordinates
  - [+ ] Finding neighbors is \(O(1)\)
  - [+ ] Easy to keep geometry manifold
  - [+ ] Supports nonmanifold geometry

- Cons:
  - [- ] Larger memory footprint
  - [- ] Hard to change connectivity with fixed indices
  - [- ] Difficult to add/remove mesh elements

\[
\begin{array}{ccccc}
\text{VERTEX} & \leftrightarrow & \text{EDGE} & \\
\text{EDGE} & \leftrightarrow & \text{FACE} & \\
\hline
\begin{array}{cccc}
v0 & v1 & v2 & v3 \\
e0 & 1 & 1 & 0 & 0 \\
e2 & 0 & 0 & 1 & 1 \\
e4 & 0 & 0 & 1 & 1 \\
e1 & 0 & 1 & 1 & 0 \\
e3 & 1 & 0 & 0 & 1 \\
e5 & 0 & 1 & 0 & 1 \\
\end{array} & \\
\begin{array}{cccc}
e0 & e1 & e2 & e3 & e4 & e5 \\
f0 & 1 & 0 & 0 & 1 & 0 & 1 \\
f2 & 1 & 1 & 1 & 0 & 0 & 0 \\
f1 & 0 & 1 & 0 & 0 & 1 & 1 \\
f3 & 0 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\end{array}
\]
Halfedge Data Structure

- Let’s store a little, but not a lot, about our neighbors:
  - Halfedge data structure added to our geometry
  - Each edge gets 2 halfedges
    - Each halfedge “glues” an edge to a face

- **Pros:**
  - [+ ] No duplicate coordinates
  - [+ ] Finding neighbors is O(1)
  - [+ ] Easy to traverse geometry
  - [+ ] Easy to change mesh connectivity
  - [+ ] Easy to add/remove mesh elements
  - [+ ] Easy to keep geometry manifold

- **Cons:**
  - [- ] Does not support nonmanifold geometry

```c
struct Halfedge
{
  Halfedge* twin;
  Halfedge* next;
  Vertex* vertex;
  Edge* edge;
  Face* face;
};
```
Halfedge Data Structure

- Makes mesh traversal easy
  - Use “twin” and “next” pointers to move around the mesh
  - Use “vertex”, “edge”, and “face” pointers to grab element

```
struct Halfedge {
    Halfedge* twin;
    Halfedge* next;
    Vertex* vertex;
    Edge* edge;
    Face* face;
};
```

**Example:** visit all vertices in a face

Halfedge* h = f->halfedge;
do {
    h = h->next;
    // do something w/ h->vertex
}while( h != f->halfedge );

**Example:** visit all neighbors of a vertex

Halfedge* h = v->halfedge;
do {
    h = h->twin->next;
}while( h != v->halfedge );

**Note:** only makes sense if mesh is manifold!
Halfedge Data Structure

- Halfedge meshes are always manifold!
- Halfedge data structures have the following constraints:

  - $h \rightarrow \text{twin} \rightarrow \text{twin} = h$ // my twin’s twin is me
  - $h \rightarrow \text{twin} \neq h$ // I am not my own twin
  - $h2 \rightarrow \text{next} = h$ // every h’s is someone’s “next”

- Keep following **next** and you’ll traverse a face
- Keep following **twin** and you’ll traverse an edge
- Keep following **next->twin** and you’ll traverse a vertex

- **Q: Why, therefore, is it impossible to encode the red figures?**
  - First shape violates first 2 conditions
  - Second shape violates 3rd condition
Connectivity vs Geometry

- Recall manifold conditions (fans not fins):
  - These conditions say nothing about vertex positions! Just connectivity

- Can have perfectly good (manifold) connectivity, even if geometry is awful
  - Can have perfectly good manifold connectivity for which any vertex positions give “bad” geometry!

- Leads to confusion when debugging:
  - Mesh looks “bad”, even though connectivity is fine
• Alpha Blending

• The Graphics Pipeline Revisited

• Introduction to Geometry
  • Implicit & Explicit Geometry
  • Manifold Geometry

• Local Geometric Operations
Edge Flip

Goal: Move edge e around faces adjacent to it:

- No elements created/destroyed, just pointer reassignment
- Flipping the same edge multiple times yields original results
Edge Flip

// collect
h = e->halfedge;
t = h->twin;
v1 = h->next->vertex;
v2 = t->next->vertex;
v3 = h->next->next->vertex;
v4 = t->next->next->vertex;
f1 = h->face;
f2 = t->face;

// disconnect
v1->halfedge = h->next;
v2->halfedge = t->next;
f1->halfedge = h;
f2->halfedge = t;

// connect
t->vertex = v3;
h->vertex = v4;
f1->halfedge = h;
f2->halfedge = t;
**Goal:** Insert edge between vertex $v$ and midpoint of edge $e$:

- Creates a new vertex, new edge, and new face
- Involves much more pointer reassignments
Edge Collapse

**Goal:** Replace edge \((c,d)\) with a single vertex \(m\):

- Deletes a vertex, (up to) 3 edges, and (up to) 2 faces
  - Depends on the degree of the original faces
Local Operations

Many other local operations you will explore in your homework...
Local Operation Tips

• Always draw out a diagram
  • We’ve given you some unlabeled diagrams
  • With pen + paper, label the elements you’ll need to collect/create

• Stage your code in the following way:
  • Create
  • Collect
  • Disconnect
  • Connect
  • Delete

• Write asserts around your code
  • Check if elements that should be deleted were deleted
  • Make sure there are no dangling references to anything that has been deleted
  • Make sure every element that you disconnected or reconnected is still valid
    • What it means for a vertex to be valid is not the same as what it means for an edge to be valid, etc.