INF3320

Computer Graphics and Discrete Geometry

The OpenGL pipeline

Christopher Dyken and Martin Reimers

12.10.2011
Pipeline operations
OpenGL fixed-function pipeline:

Implements a fixed sequence of operations:

- transformation of vertices
- lighting calculations for vertices
- combining vertices to primitives
- rasterization
- texturing of fragments
- combining fragments to framebuffer values

State-machine: Fixed operations can be turned on or off

Green = programmable, yellow = configurable, blue = fixed
Today

OpenGL fixed function functionality

- combining vertices to primitives
- rasterization
- coloring of fragments
- combining fragments with framebuffer values

Green = programmable, yellow = configurable, blue = fixed
The OpenGL pipeline

Real Time Rendering:
- The Graphics Rendering Pipeline (Chapter 2)
- The Graphics Processing Unit (GPU) (Chapter 3)
- Buffers and Buffering (Chapter 18.1)
- Perspective Correct Interpolation (Chapter 18.2)

The Red Book:
- Blending, Antialiasing, and Polygon Offset (Chapter 6)
- The Framebuffer (Chapter 10)
- Order of Operations (Appendix A)
Primitive assembly / Triangle setup
Primitive assembly and culling

- Primitive assembly combines transformed vertices into
  - points,
  - lines,
  - triangles
  according to `glBegin-glEnd` state.

- Triangles are classified\(^1\) as **front-** or **backfacing**.

- When culling is enabled\(^2\), a triangle is discarded if
  - it is backfacing and `glCullFace(GL_BACK)` is set, or
  - it is frontfacing and `glCullFace(GL_FRONT)` is set.

⇒ If an object is **closed**, we can never see the back-side of the triangles, so why process them further?

\(^1\)Check the sign of the 2D-area of the triangle.  
\(^2\)`glEnable(GL_CULL_FACE)`
Clipping
Hyperplanes and half-spaces

- A hyperplane divides a space into two, a pos and neg side:
  - In $\mathbb{R}$ a point is a hyperplane,
  - in $\mathbb{R}^2$ a line is a hyperplane, and
  - in $\mathbb{R}^3$ a plane is a hyperplane.

- In $\mathbb{R}^d$, a hyperplane is defined implicitly by $a_1, \ldots, a_{d+1}$:

  \[
  \sum_{i=1}^{d} a_i p_i + a_{d+1} > 0 \text{ implies that } p \text{ is in the positive halfspace,}
  \]

  \[
  \sum_{i=1}^{d} a_i p_i + a_{d+1} < 0 \text{ implies that } p \text{ is in the negative halfspace,}
  \]

  \[
  \sum_{i=1}^{d} a_i p_i + a_{d+1} = 0 \text{ implies that } p \text{ is on the hyperplane}
  \]
Recall that the point \( p = (x, y, z, w) \) is on the homogeneous plane \( m = [a \ b \ c \ d] \) if

\[
\begin{bmatrix} a & b & c & d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = ax + by + cz + dw = 0.
\]

We can find the intersection of the plane and a line segment \( p_1 \rightarrow p_2 \) by inserting into the equation above,

\[
m \cdot p = m \cdot ((1-t)p_1 + tp_2) = 0,
\]

by solving for \( t \)...
\[ t = -\frac{mp_1}{m(p_2 - p_1)} \] gives the parameter point of the intersection.

If \( p_1 \) and \( p_2 \) have associated data like color \( c_1 \) and \( c_2 \), the color \( c \) at the intersection point can be linearly interpolated as

\[ c = (1 - t)c_1 + tc_2. \]
Clipping line segments in the plane

Assume we want to clip against a rectangle $[-w, w][-h, h]$ in $\mathbb{R}^2$:

- rectangle is the intersection of the four half-planes,

\[ m_1 = \begin{bmatrix} 1 & 0 & w \end{bmatrix} \quad x \geq -w, \]
\[ m_2 = \begin{bmatrix} -1 & 0 & w \end{bmatrix} \quad x \leq w, \]
\[ m_3 = \begin{bmatrix} 0 & 1 & h \end{bmatrix} \quad y \geq -h, \text{ and} \]
\[ m_4 = \begin{bmatrix} 0 & -1 & h \end{bmatrix} \quad y \leq h. \]

$\Rightarrow$ A point $p = \begin{bmatrix} x & y & 1 \end{bmatrix}^T$ is inside the rectangle, if

\[ m_i p \geq 0, \quad \text{for } i = 1, 2, 3, 4. \]
We then clip in the following way:

For each hyperplane $i = 1, 2, 3, 4$, clip the line segment:

1. If both end-points is outside the half-space $\implies$ reject it. $\implies$ reject the line segment.
2. If both end-points is inside the half-space $\implies$ accept it. $\implies$ accept the line segment.
3. If one end-point is inside, and one end-point is outside $\implies$ define a new line segment between the end-point inside and the intersection point.
The intersection of two convex sets is convex. Half-spaces are convex, and therefore, the clipped result is convex.

A polygon is defined as a loop of vertices

\[ \mathbf{p}_1 \rightarrow \mathbf{p}_2 \rightarrow \cdots \rightarrow \mathbf{p}_n \rightarrow \mathbf{p}_1 \]

Then, for each half-space
1. Classify each vertex as inside or outside the half-space.
2. for each pair of adjacent vertices where one vertex is inside and one is outside, insert the intersection point between.
3. Remove the vertices classified as outside

The remaining loop represents the clipped polygon.
Clipping stage in OpenGL

The geometry is clipped against the 6 planes of the view frustum

Additionally, >6 more planes can be defined, e.g. for cutaway view:

```gl
    glEnable(GL_CLIP_PLANEi)
    glClipPlane(plane-id, [A,B,C,D])
```

- vertices with eye-space coords \((x, y, z, w)\) such that

  \[
  (A, B, C, D)M^{-1}(x, y, z, w) < 0
  \]

  are clipped \((M = \text{modelview matrix})\)
Clipping stage in OpenGL
Rasterization
Rasterization

Convert primitives to fragments (which contributes to pixel color)
Rasterization and the fragment pipeline

> Rasterization converts **primitives** into **fragments**\(^3\).

> Fragments are then subjected to **fragment pipeline** and then sent to buffer operations.

\(^3\)A pixel and associated data before it is written to the framebuffer
Which fragments do a triangle produce?

- All fragments whose centre is inside the triangle is produced.
- A tie-breaking rule is used when the centre is on the boundary.
  
  A fragment belongs only to one of two adjacent triangles.
Fragment invariance

Fragments hold the following invariance rules:

▶ Given the primitive \( p \)
  ▶ make a new primitive \( p' \) by offsetting the coordinates with the integers \( x, y \).
  ▶ Assume that neither \( p \) or \( p' \) is clipped.

\[ \implies p \text{ and } p' \text{ produces the exact same fragments, except that fragment } f' \text{ of } p' \text{ is shifted } x, y \text{ compared to fragment } f \text{ of } p. \]

▶ If you render two primitives with identical vertex coordinates, they produce exactly the same fragments.

Why? This is of major importance in multi-pass techniques where one renders the same geometry more than once.
Interpolating fragment data over primitives

Need to interpolate vertex data in fragments (position, color etc.)

**Barycentric coordinates**

The barycentric coordinates $(\alpha_1, \alpha_2, \alpha_3)$ of $p$ w.r.t the triangle $[p_1, p_2, p_3]$ are

\[
\alpha_1 = \frac{\text{area}(p, p_2, p_3)}{\text{area}(p_1, p_2, p_3)}, \quad \alpha_2 = \frac{\text{area}(p, p_1, p_3)}{\text{area}(p_1, p_2, p_3)}, \quad \alpha_3 = \frac{\text{area}(p, p_1, p_2)}{\text{area}(p_1, p_2, p_3)}
\]

They are the unique **convex weights**, i.e.

\[
\alpha_1, \alpha_2, \alpha_3 \geq 0, \quad \alpha_1 + \alpha_2 + \alpha_3 = 1,
\]

s.t $p = \alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3$

Yields a natural way to linearly interpolate data over triangles: e.g. color

\[
c(p) = \alpha_1 c_1 + \alpha_2 c_2 + \alpha_3 c_3.
\]
Linear interpolation

Let \((\alpha_1, \alpha_2, \alpha_3)\) be the barycentric coords of the fragment midpoint wrt. the screen-space coordinates of \(T = [p_1, p_2, p_3]\).

**Interpolating linearly in screen space**

Linear interpolation of corner colors \(c_1, c_2, c_3\)

\[
c = \alpha_1 c_1 + \alpha_2 c_2 + \alpha_3 c_3
\]

Looks wrong since perspective distortion is lost!

Must take depth \(d\) into account! In 2D:
Linear interpolation

Let \((\alpha_1, \alpha_2, \alpha_3)\) be the barycentric coords of the fragment midpoint wrt. the screen-space coordinates of \(T = [p_1, p_2, p_3]\).

Interpolating linearly in screen space

Linear interpolation of corner colors \(c_1, c_2, c_3\)

\[
c = \alpha_1 c_1 + \alpha_2 c_2 + \alpha_3 c_3
\]

Looks wrong since perspective distortion is lost!

Must take depth \(d\) into account! In 2D:

\[
p_1 = \frac{v_1}{d_1}
\]

\[
p_2 = \frac{v_2}{d_2}
\]

Assume \(p = \alpha_1 p_1 + \alpha_2 p_2\). Then

\[
v = \frac{\alpha_1 \frac{v_1}{d_1} + \alpha_2 \frac{v_2}{d_2}}{\frac{\alpha_1}{d_1} + \frac{\alpha_2}{d_2}} = \tilde{\alpha}_1 v_1 + \tilde{\alpha}_2 v_2
\]

i.e. \(\tilde{\alpha}_1, \tilde{\alpha}_2\) are coordinates of \(v\) wrt \(v_1, v_2\)
Perspective correct interpolation

Interpolating linearly in **object** space

Perspective corrected interpolation is given by

\[ c = \frac{\alpha_1}{d_1} c_1 + \frac{\alpha_2}{d_2} c_2 + \frac{\alpha_3}{d_3} c_3 \]

where \( d_1, d_2, d_3 \) are the depths of the vertices

Perspective correct interpolation!

Turn on with \texttt{glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)}

All modern graphics cards support perspective correct interpolation

Easy GLSL shader support
The effect of perspective correct interpolation

Screen space interpolation of texture coordinates

Flat

Affine

Correct
Fixed fragment pipeline

Fragments (position w/ data) are submitted to **fragment pipeline**:

▶ If texturing is enabled
  ➞ Use texcoords and do a texture lookup
  ➞ modify the fragment color.

▶ If secondary color\(^4\)
  ➞ add secondary colour to the fragment’s color

▶ If fog is enabled,
  ➞ use depth to control a blend between the fragment color and a fog colour.

▶ The fragment is then sent to framebuffer operations.

\(^4\)If `glEnable(GL_SEPARATE_SPECULAR_COLOR)` is enabled, the specular contribution from the lighting calculations are maintained in a **secondary color** which is not tampered with by texturing.
The framebuffer
A framebuffer is a set of logical buffers

- color buffers (front/back, left/right, + auxiliary, typically 16–32 bits)
- depth buffer (typically 24–32 bits) (store depth value per pixel)
- stencil buffer (typically 0–8 bits) (used to control drawing/masks)
- accumulation buffer (precision like color buffer or higher) (used e.g. for combining several images), . . .

A fragment/pixel is related to the contents of all logical buffers at its position
Basic Buffer operations

The buffers are read from and/or written to in fragment-pipeline

Clearing

- `glClear[Color|Depth|Stencil|Accum]` sets the clear value, e.g. `glClearDepth(1.0f)`
- `glClear(bit-mask)` clear the chosen buffers, e.g. `glClear(GL_DEPTH_BUFFER_BIT|GL_COLOR_BUFFER_BIT)`

Read and Write

- `glDrawBuffer` select color buffers for writing/clearing
- `glReadBuffer` select src for reading (e.g. `glReadPixels`)

Fixed-function pipeline: each output-buffer recieve the same output
Programmable pipeline: output-buffers can get individual output
Write masks

specifies which of current framebuffer’s logical buffers are updated:

   glColorMask(bool,bool,bool,bool)
   specifies if the R,G,B, or A channels should be updated.

   glDepthMask(bool)
   specifies if the depth buffer should be updated\(^1\).

   glStencilMask(bitmask)
   specifies which of the stencil-buffer bit-planes that should be updated.
   (Comes in a separate front/back mode too)

---

Render only to the stencil buffer

turn off the color and depth masks and enable the stencil mask.

---

\(^1\) Use `glEnable/glDisable` with `GL_DEPTH_TEST` to control the test itself.
Per-fragment operations
All fragments are subjected to a series of tests:

Fragment and associated data

- ownership test
- scissor test
- alpha test (RGBA)
- stencil test

write

framebuffer

read / update

Logical operations
- Dithering
- blending (RGBA)
- depth test

If all tests succeed \(\Rightarrow\) fragment is written to the framebuffer

If any test fail \(\Rightarrow\) fragment is discarded.
Fragment tests

Pixel ownership test
The OS discard fragments at pixel locations not owned by the context.

Scissor test (`glScissor(x,y,w,h)`)  
All fragments outside a specified rectangle are discarded.

Alpha test (`glAlphaFunc(func,ref)`) (deprecated from GL 3.0)  
Compare (func) the fragment’s alpha value with a reference value (ref)  
Let us e.g. render opaque and translucent objects, billboarding...
The stencil test

Stencil test

Compare value in the stencil buffer with a reference value.

\texttt{glStencilFunc(func,ref,mask)} comparison function 
(never,always,\(<,\leq,\neq,\ldots\)), reference value, and a bitmask for both reference and stencil buffer value

Stencil operation

The stencil buffer is updated from the result of the stencil and depth-tests.

\texttt{glStencilOp} specifies how the stencil buffer is updated when:

- the stencil-test fails
- the stencil-test passes, but dept-test fail
- both the stencil-test and the depth-test passes

\texttt{glEnable/Disable( GL_STENCIL_TEST )} turns the test on/off.

Typically used for disabling rendering to a part of the screen
The depth test

The depth-buffer (z-buffer) algorithm

clear by setting all depth-buffer elements to far.

for each fragment:
  ▶ if fragment’s depth < depth-buffer depth
      ⇒ store fragment and update depth-buffer
  ▶ if fragment’s depth ≥ depth-buffer depth
      ⇒ discard fragment

result is that only the nearest fragment is stored for any sequence of fragments.

glEnable/Disable( GL_DEPTH_TEST ) turns the test on/off.
glDepthFunc specifies the depth test (GL_NEVER, GL_LESS, ...)
The Z-buffer

A simple three dimensional scene

Z-buffer representation
What is stored in the depth-buffer?

\[
depth(z) = \lfloor 2^{\text{depth bits}} \left( \frac{z_{\text{far}}}{z_{\text{far}} - z_{\text{near}}} - \frac{1}{z} \frac{z_{\text{far}} z_{\text{near}}}{z_{\text{far}} - z_{\text{near}}} \right) \rfloor
\]

depth is an \( n \)-bit integer proportional to \( \frac{1}{z} \) (i.e. non-linear).

much precision around the near plane

little precision around the far plane

**Increase depth precision by pushing near plane as far away as possible!**

The far plane also influences precision, but to a much lesser extent.
Z-fighting is the result of too little z-buffer precision

Two fragments of different z-values get the same depth value, and the algorithm can’t tell which is in front of the other.

This results in an ugly pattern where some fragments of one primitive appears in front of the other while others do not.
Stencil/depth test applications

DOOM3 shadows: count back and front facing fragments passing depth test
The merging stage

Incoming (src) fragments are merged into the framebuffer (dst):

- Depth/z-test: replace dst fragment if src is closer
- Logical operations: AND, XOR, ...
- Blending: combine src and dst

Logical operations, `glLogicOp(OP)`

\[ dst = src \ OP \ dst \]

where OP is a logical operation, e.g. AND, OR,...
Blending

The incoming fragment’s is combined with the color already in the framebuffer.

\[ \text{src} = F_s \text{src \ OP \ F}_d \text{dst} \]

where src and dst are RGBA, F’s are blend-factors and OP is e.g. add-operator

Set up with

- `gl[Enable|Disable](GL_BLEND)` - enables blending
- `glBlendFunc(F_s, F_d)` - set RGBA blend factors
- `glBlendFuncSeparate(F_s, F_d, F_s^\alpha, F_d^\alpha)` - separate RGB and \( \alpha \) blend factors
- `glBlendEquation(OP)` - OP can be e.g. add, subtract, min, max, etc.

The factors \( F \) can be: 0, 1, \( \alpha \), src, dst, ...
Blending is most often based on the alpha channel

- Transparent objects, render back to front
- Fog
- Billboardging
- Anti-aliasing
Transparency

Blend color of transparent material and background

Important: Render back to front (oblique objects first)!
Fog

Popular, cheap effect for depth cues:

\[ c = f c_f + (1 - f) c_s \]

where \( c_f \) and \( c_s \) are fog and surface color, and \( f \) can be

- **linear**: \( f(z) = \frac{z_{end} - z}{z_{end} - z_{start}} \)
- **exp**: \( f(z) = e^{-d_f z} \)
- **exp2**: \( f(z) = e^{-(d_f z)^2} \)
Fog in OpenGL

GLfloat fogColor[4] = {0.5, 0.5, 0.5, 1.0};

glEnable (GL_DEPTH_TEST); //enable depth testing
glEnable (GL_FOG);
glFogi (GL_FOG_MODE, GL_EXP2);

glFogfv (GL_FOG_COLOR, fogColor);
glFogf (GL_FOG_DENSITY, 0.3);
glHint (GL_FOG_HINT, GL_NICEST);  // pr pixel calculation
Accumulation buffer

A color buffer that can accumulate several renderings, like multiple exposures

- `glAccum(op,val)`, e.g. `glAccum(GL_ACCUM,0.1)`
- read from readable buffer (`glReadBuffer`)
- After accumulation, `glAccum(GL_RETURN,1)` writes back to write to output-buffers

Applications: accumulate perturbed images

- Antialiasing
- Depth of field
- Motion blur
Main points today: opengl pipeline, blending, frame-buffer

Fixed function pixel operations are somewhat obsolete..

You will see why next time - programmable pipeline
Gamma Correction - the problem

Perceived intensity is usually not a linear function of light intensity
- eyes are more sensitive to variation in low intensity
- hardware handles signal strength (voltage) differently

Most computations in gfx are based on linearity, e.g. adding contributions from multiple lights

This gives wrong results in many cases, e.g. in antialiasing
Gamma Correction - the solution

In order to represent color correctly we modify signal by

\[ V_{out} = V_{in}^\gamma \]

to account for the hardware specific pixel intensity to emitted/percieved intensity.

\(\gamma\)-correction is important for portability, image quality, texturing and interpolation. Handled by modern GPUs, or using shaders.