lecture 7
- graphics pipeline (overview)
- hidden surface removal
  - object vs image order
  - back face culling
  - depth buffer (z buffer)
  - painters algorithm
  - ray casting

graphics pipeline
(lectures 1-6)
- clip coordinates
- coordinate transforms
- clipping
- rasterization

OpenGL pipeline (on the graphics card)
- vertex processing
- "primitive assembly" and clipping
- rasterization
- fragment processing

A "fragment" is a potential pixel and the data needed to color it, including depth, surface normal.

classic vs. modern OpenGL

"Particle systems" (vertex processing)
e.g. Fire, explosions, smoke, fog, ...
Calculate geometric transforms on vertices/primitives.
Calculate (time varying) "color" of vertices, too!

"Primitive Assembly" and Clipping

Suppose you want to make a water wave animation. The surface is a set of triangles, made from vertices
\[
\{(x, \text{height}(x,z,t)), z\}
\]
height() is a little program -- a "vertex shader"
e.g. a sine wave.

In classic OpenGL, the CPU calculates \(\text{height}(x, z, t)\) for each \(x, z\) and then calls \text{glVertex}().

In modern OpenGL, \(\text{height}(x, z, t)\) is computed using a "vertex shader", on the graphics card and in parallel for different \((x, z)\) and \(t\).

"Fixed function"  \hspace{1em}  "Fragment shader" (programmable)

OpenGL 1.0  \hspace{1em}  Modern OpenGL

"Particle systems" (vertex processing)

Calculate geometric transforms on vertices/primitives.
Calculate (time varying) "color" of vertices, too!

Careful: no pixels yet !

Fragment processing

A fragment is a potential pixel. It has an \((x, y)\) coordinate, and information about depth, color, ...

We will discuss fragment processing later in the course.
Part 2 of the course starts here.

Visibility, geometry modelling

- hidden surface removal
- back face culling, depth buffer, painter, ray casting
- efficient ray casting
- bounding volumes, octrees, BSP tree
- object hierarchies
- OpenGL, transformation stack, scene graph
- object hierarchies
- view, clipping, C systems, plant models
- smooth curves and surfaces
- visible: Hermite curves & Catmull-Rom splines, bicubic surfaces
- meshes
  - level of detail, edge collapse

lecture 7

- graphics pipeline (overview)
- hidden surface removal:
  - the problem of deciding which polygon/object is visible at each pixel.
    - object vs image order
    - back face culling
    - depth buffer (z buffer)
    - painters algorithm
    - ray casting

"object order" methods

for each object
  for each pixel
    decide if object is visible at that pixel

"image order" methods

for each pixel
  for each object
    decide if object is visible at that pixel

Back face culling (object order)

In OpenGL, the **front face** of a polygon is defined (by default) as the side where vertices would be ordered counter clockwise, i.e. if a viewer were on that side.

```cpp
glFrontFace(GL_CCW) // default
```

Choose ordering of vertices for each face as shown so that front faces are seen by a viewer.

```cpp
glFrontFace(GL_CW) // override default
```

The normal comes out of the front face.
(The front face has an "outward pointing normal" and a back face has an "inward pointing normal".)

[WARNING: In OpenGL, surface normals also can be explicitly defined defined at vertices. (We will see later why.) But these normals have nothing to do with front and back faces as just defined.]

"Back Face Culling"

= don't draw the back faces!

For a solid object, back faces shouldn't be visible because they are hidden by the front faces.
If back face culling is "enabled", then the back face is not drawn.

In A1, we used `glDisable(GL_CULL_FACE)`.

When is a polygon back vs. front face?

*In camera coordinates,* it is subtle. You can't just look at the sign of the z coordinate.

Use the sign of the dot product of the outward facing normal and the vector from the viewer to any vertex on the polygon.

Q: Where is back face culling done?

A: In OpenGL it is done by rasterizer. But in principle, it can be done before entering the pipeline!

In OpenGL, back face culling is done by the rasterizer, hence it is done in *normalized device coordinates.* Check the sign of the normal's z coordinate.

Rule:

- \( \hat{n} \cdot \hat{z} > 0 \) \quad \text{back face}
- \( \hat{n} \cdot \hat{z} < 0 \) \quad \text{front face}

Hidden surface removal algorithm (depth buffer)  
Catmull 1974

for each polygon P:
  for each pixel \((x,y)\) in \(P\)'s image projection
    if \(P\)'s depth at \((x,y)\) is less than \(z\text{buffer}(x,y)\)
      update \(z\text{buffer}(x,y)\)
      compute color of \(P\) at \((x,y)\) and replace the current color with the new color

Pseudocode...

for each pixel \((x,y)\):
  // initialization
  \(z\text{buffer}(x,y) = 1\)
  RGB\((x,y)\) = background color

for each polygon:
  for each pixel \((x,y)\) in the image projection of polygon
    \(z := Ax + By + C\)
  // equation of polygon's plane in SCREEN coordinates
  if \(z < z\text{buffer}(x,y)\):
    \(z\text{buffer}(x,y) := z\)
    compute \(RGB(x,y)\)

Q: Where is the depth buffer algorithm here?

A: It is done by the rasterizer. If polygon fails the depth test at \((x,y)\), then fragment never gets generated.
Recall last lecture: "window to viewport" transformation

Recall \texttt{glFrustum}( _, _, _, _, _, near, far) where near and far are float or double.

Q: Why not set:

\begin{align*}
near &= 0.00000...1 \\
near &= 2^{1023} \\
\end{align*}

A: You would divide the huge \([\text{near}, \text{far}]\) interval into \(2^{24}\) depth intervals/bins. Most points will fall in one of these intervals \(\rightarrow\) useless because they all have same (quantized/rounded) depth.

Depth buffer typically holds fixed precision (e.g. 24 bits), not float.

i.e. \(z\) in \([0, 1]\) partitioned into bins of size \(1/2^{24}\).

Q: Why?

A: Floating point is useful for representing tiny and huge numbers, but that's in appropriate here.

There is also a depth component to the mapping.

\( (x, y, z) \) in \([-1, 1] \times [-1, 1] \times [-1, 1] \)

Recall \( (x, y) \) in \([0, \text{width}] \times [0, \text{height}] \)

Painters Algorithm [Newell et al 1972]

\begin{itemize}
  \item Sort polygons in depth. (Arbitrarily) use the farthest vertex in each polygon as the sorting key.
  \item e.g. 26 polygons \([A, B, C, \ldots, P, Q, \ldots, Z]\)
  \item Then draw polygons from "farthest" to "nearest" (back to front).
\end{itemize}

However, that doesn't always work. Why not?

Typical failure of method on previous slide:

\begin{itemize}
  \item desired \( \z = 1 \)
  \item (naive) Painter \( \z = 1, \z = 3, \z = 1 \)
  \item Solution? Swap order of \(P\) and \(Q\) in the list? (That can create other problems.)
\end{itemize}

A related common example of failure

\begin{itemize}
  \item Problem (cut \(Q\) into \(Q_1, Q_2\))
  \item Solution
\end{itemize}
Q: When is it safe to draw Q before P?
A: when \( Q \) does not occlude \( P \).

Any of the following:
- all \( Q \)'s vertices are farther than all \( P \)'s vertices
- the \( x \)-range of \( P \) and \( Q \) do not overlap
- the \( y \)-range of \( P \) and \( Q \) do not overlap
- ... (previous slides problems not covered yet)

SLIDE ADDED (morning after)
I have not given the full Painter's Algorithm.
I have not shown how to deal with every case.
The full algorithm has more details that I want to cover.

But I think you get the main ideas. And its enough for me to motivate BST trees next lecture.

Ray Casting (an image order method)

for each pixel \((x,y)\) {
    for each polygon{
        check if the pixel lies in the image of that polygon and if the depth is smallest seen so far at that pixel.
    }
    draw the color of the nearest polygon at that pixel
}

Ray Casting

for each pixel \((x,y)\)

\[
\begin{align*}
\text{zMin} &= 1 \quad \text{// initialized to max value} \\
\text{for each polygon}{ \quad \left\{ \\
\quad \text{if} \ (x,y) \\text{lies in image projection of this polygon}{ \quad \left\{ \\
\quad \quad z &= Ax + By + C \quad \text{// screen coordinates} \\
\quad \quad \text{if} \ z < \text{zMin} \quad \text{zMin} := z \\
\quad \quad \quad \text{pixel}(x,y).poly = \text{polygon} \\
\quad \quad \right\} \\
\quad \right\} \\
\quad \text{RGB}(x,y) &= \ldots?\ldots \quad \text{// only draw once per pixel}
\end{align*}
\]

No z-buffer needed.
For each pixel, only choose its color once.

Need to determine if a pixel lies in a polygon.
(Details omitted.)
More general ray casting (used next lecture)

Does ray from p₀ through p₁ intersect the polygon?

Typically p₀ is the camera position. But not necessarily...

Trick: project the polygon and the intersection point with plane orthographically into a canonical plane and use the 2D "point in polygon" solution.

How to decide if the intersection point lies in the polygon?

Does ray intersect this quadric?

-[ x, y, z, 1 ] \ Q \ [ x, y, z, 1 ]^T = 0

(x(t), y(t), z(t)) = p₀ + (p₁ - p₀) t

Substitute the ray into the quadric gives a second order equation:

α t^2 + β t + c = 0

Solve for t.

Gives two solutions. What are the possibilities?

Two real roots.
If at least one is positive, then take the smaller positive one.

Two real roots (identical).

Both roots are complex.

Announcement

A1 was posted on Friday. It is due next Monday.

Example of solution (executable):

TA office hours this week.

DISCUSSION BOARD SETTINGS.
Please uncheck: "Include original post in reply"