DH2323 DGI17

INTRODUCTION TO COMPUTER GRAPHICS AND INTERACTION

RASTERISED RENDERING

Christopher Peters
CST, KTH Royal Institute of Technology, Sweden

chpeters@kth.se
http://kth.academia.edu/ChristopherEdwardPeters

Based on DGI12 notes by Carl Henrik Ek
The Rendering Equation

\[ L_o(x, \omega_o, \lambda, t) = L_e(x, \omega_o, \lambda, t) + \int_{\Omega} f_r(x, \omega_i, \omega_o, \lambda, t) L_i(x, \omega_i, \lambda, t) (\omega_i \cdot n) \, d\omega_i \]

Describes:
Total amount of light emitted from a point \( x \) along a specific viewing direction

Given:
- Incoming light function
- BRDF

Basis:
- Law of conservation of energy

BRDF
Account for angle w.r.t. light
Incoming radiance
Hemisphere containing all \( \omega_i \)
Global Illumination

Ray tracing:
– Good for specular
– Bad for diffuse

Radiosity:
– Good for diffuse
– Bad for specular

Hybrid techniques
Caustics

- Curved regions of bright reflected or refracted light
Sub-surface scattering

- Light bouncing around inside material before exiting
Ray tracing

Pixel order rendering technique
- Trace at least one ray through each image pixel
- Maintains primitives in geometric scene model
- Queries this for each ray
- Determine which primitive is visible for each pixel

Geometry queries can have high cost
Rasterisation

Scanline: object order based

*Fragments*
- Data for single pixel
- Frame buffer

Handle occlusion using depth buffer
- Later details (more specifically, *fragments*) overwrite earlier ones if closer to camera

Shade based on vertices and interpolate
- See lighting and shading lecture
Rasterisation

Process of converting geometry into a raster image (series of pixels)

- 3D scene converted into 2D image
- Polygons
- ...composed of triangles
- ...composed of vertices
Rasterisation

Rasteriser takes stream of vertices
Project them onto the 2D surface of the screen
Fill in the interior of the 2D triangles

Core concepts:
– Geometry and transformations
– Projection
– Clipping
– Scanline Conversion
Geometry Transformations

Matrix multiplication
Translation, scaling, rotation, projection
Familiar?
Transformation Stack

*Stack* of transforms (i.e. matrices)
- Push and pop
Position stream of input vertices
Incoming vertices transformed according to the transformation stack
Remember: local coordinate marker idea
Projection

Remove depth
  – Convert 3D geometry to flat 2D representation
  – Do so for each vertex of each polygon

Orthographic projection
  – Simply remove z coordinate
  – Viewing volume is a cube

Perspective projection
  – Single point of projection (focal point)
  – Viewing volume is a pyramid
Parallel Projections

axonometric

orthographic

oblique
Orthographic Projection

From OpenGL Programming Guide
Perspective Projection

From OpenGL Programming Guide
Example

Perspective projection (P)  Orthographic projection (O)

Nicolas P. Rougier, ERSF Code camp
Projection
Camera Specification

Parameters
Centre of projection (COP)
Field of view (FOV)
Projection direction
Up direction
Outcomes

Large FOV

Small FOV
Clipping

Projected locations may be outside the viewing window
Truncate triangles to fit them inside the viewing area
– e.g. Sutherland-Hodgeman algorithm

Real-time Rendering, Akenine-Moller, Haines and Hoffman
Scan Conversion

Fill interior of triangles in image plane
Use *scanline fill algorithm* to fill polygons
Framebuffer
Line Drawing

A line usually defined as infinitely thin

How to display using pixels?

Fixed and finite area

Choose pixels that best represent the line

Different algorithms, providing different results:

mid-point

neighbourhood

weighted area
Line Drawing

Bresenham's line algorithm

dx = x_end - x_start
dy = y_end - y_start
d = 2 * dy - dx
x = x_start
y = y_start
while x < x_end
    if d <= 0 then
        d = d + (2 * dy)
        x = x + 1
    else
        d = d + 2 * (dy - dx)
        x = x + 1
        y = y + 1
    endif
    SetPixel(x,y)
endwhile
Polygon Filling

Fill surface
Triangles
Interpolation
Compute edges
Fixed Function Pipeline (FFP)

**Beware:** what follows is mostly deprecated

- Not programmable
  - Transform and rasterisation operations hardwired
- *Immediate mode* OpenGL
- Obsolete

Compare to modern *programmable pipeline*

- Vertex, pixel, etc shaders

**Recommendation:**

- Start with FFP to gain understanding of basics
- Learn programmable pipeline after
Graphics Pipeline Architecture

Can divide pipeline into three conceptual stages: 
*Application* (input, animations, think SDL) 
*Geometry* (transforms, projections, lighting) 
*Rasteriser* (draw image as pixels)

These define the core structure of the pipeline
Geometry Stage

Responsible for polygon and vertex operations
Consists of five sub-stages:

- Model and View Transform
- Lighting and Shading
- Projection
- Clipping
- Screen Mapping
What is OpenGL?

Software interface to graphics hardware
Commands for interactive three-dimensional graphics
Hardware independent interface
Drawing operations performed by underlying system and hardware

C/C++ Code → OpenGL Application → OpenGL → GLUT → GLU → Operating System → Xlib, Xtk → Graphics Hardware
OpenGL Conventions

All function names begin with **gl**, **glu** or **glut**

- `glVertex(…)`
- `gluSphere(…)`
- `glutMouseFunc(…)`

Constants begin with **GL_**, **GLU_** or **GLUT_**

- `GL_FLOAT`

Function names show parameter types

- `glVertex2i(1, 3)`
- `glVertex3f(1.0, 3.0, 2.5)`
- `glVertex4fv(array_of_4_floats)`
OpenGL Primitives

Basic building blocks
Line based primitives, and polygon based primitives:

Line

Polygon

GL_POINTS   GL_LINES   GL_LINE_STRIP   GL_LINE_LOOP

GL_TRIANGLES   GL_QUADS   GL_POLYGON   GL_TRIANGLE_STRIP
Some OpenGL Code

To create a red polygon with 3 vertices:

```c
glColor3f(1.0, 0.0, 0.0);
glBegin(GL_POLYGON);
  glVertex3f(0.0, 0.0, 3.0);
  glVertex3f(1.0, 0.0, 3.0);
  glVertex3f(1.0, 1.0, 3.0);
  glVertex3f(1.0, 1.0, 3.0);
  glVertex3f(0.0, 0.0, 3.0);
glEnd();
```

`glBegin` defines the start of a new geometric primitive:
3D vertices are defined using `glVertex3f`
The colour is defined using `glColor3f`
  – Colours remain selected until changed
  – OpenGL operates as a state machine
FreeGLUT

High level library for OpenGL like SDL

GLUT works using an *event loop*:

```c
while (GLUT is running)
    e = get_next_event()
    if (e == mouse moved)
        run the mouse_moved function provided by the user
    if (e == redraw)
        run the draw function provided by the user
    etc, etc
```

```c
// Initialize GLUT
glutInit(&argc, argv);
// Request RGBA, depth testing and double buffering
glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
// Create window
glutCreateWindow("RGSquare Application");
// Request window size
glutReshapeWindow(400, 400);
```

RGB Colour, depth testing and double buffering

Window title

Request for window size (not necessarily accepted)
Glut Callback Functions

Some of the callback registration functions:

- `void reshape(int w, int h) {...}
- `void keyhit(unsigned char c, int x, int y) {...}
- `void idle(void) {...
- `void mouse(int button, int state, int x, int y) {...

```
glutReshapeFunc(reshape);  window resized
glutKeyboardFunc(keyhit);   key hit
glutIdleFunc(idle);         system idle
glutDisplayFunc(draw);      window exposed
glutMotionFunc(motion);     mouse moved
glutMouseFunc(mouse);       mouse button hit
glutVisibilityFunc(visibility);  window (de)iconified
glutTimerFunc(timer);       timer elapsed
```

`glutMainLoop();`  Begin event loop
Some Glut Objects

```c
void glutSolidTorus(double inner_radius,
                     double outer_radius, int nsides, int rings);
```

```c
glutWireTorus(0.3, 1.5, 20, 20);
```

```c
glutSolidTorus(0.3, 1.5, 20, 20);
```

```c
glutSolidTeapot(1.0);
```

```c
glutSolidDodecahedron();
```
Matrices in OGL

To initialise a matrix in OpenGL, use

```c
glLoadIdentity();
```

Clears the currently selected OpenGL matrix (projection, transform, texture) to the identity matrix

*Note: always be aware of what the matrix mode is activated!*

To select a matrix as the current matrix, use

```c
glMatrixMode(mode);
```

*Mode* can be one of:

- GL_PROJECTION
- GL_MODELVIEW
- GL_TEXTURE
Perspective Projection

\[ \text{glFrustum}(\text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}, \text{zmin}, \text{zmax}); \]

- All points on the line defined by
  - \((\text{xmin}, \text{ymin}, -\text{zmin})\) and COP are mapped onto the lower left point on the viewport
  - \((\text{xmax}, \text{ymax}, -\text{zmin})\) and COP are mapped onto the upper right point on the viewport

Easier:

- \(\text{gluPerspective}(\text{fov}, \text{aspect}, \text{near}, \text{far});\)
The Viewport

\[ \text{glViewport}(x, y, \text{width}, \text{height}); \]

\((x, y)\) is the location of the **bottom left** of the viewport within the window

- \((0,0)\) is the bottom left hand side of the window
- \(\text{Width, height}\) specify the dimension in pixels of the viewport
Useful for doing this...

3DS Max, 3ptools Lite, polycount
Positioning the Camera

```
gluLookAt(eye pos x, eye pos y, eye pos z, look at x, look at y, look at z, up x, up y, up z);
```

- Position camera at `eye pos (x,y,z)`
- Make camera viewing direction `look at (x,y,z)`
- Set camera up vector to `up (x,y,z)`
Projection and Camera

The following code will:
Create a viewport the size of the entire window
Create perspective projection with:
70 degree fovy, aspect ratio of 1, near plane at 1 and far plane at 50
Position the camera at (0.0,0.0,5.0) looking at (0.0,0.0,0.0)

```c
glViewport(0,0,window_width, window_height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(70.0, 1.0, 1, 50);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0);
```
Transformations in OpenGL

\texttt{glTranslate(dx, dy, dz);} \\
Translates by a \texttt{displacement} \((dx, dy, dz)\) \\
Concatenates the specified translation to the current model-view matrix \\
Primitives drawn after this are modified by the specified translation

\texttt{glRotatef(angle, vx, vy, vz);} \\
Rotates around the axis \((vx, vy, vz)\) by \texttt{angle} degrees \\
Concatenates the specified rotate to the current model-view matrix \\
Primitives drawn after this are modified by the specified rotation
Pushing and Popping

OpenGL allows us to push and pop matrices on a stack

- Remember the transform stack
- `glPushMatrix();`
- `glPopMatrix();`

Pushing: save current local coordinate marker
Popping: retrieve/load previous local coordinate marker

There should always be the same number of matrix push operations as matrix pops

*Why?*
Example code

In order to draw two squares in different world-space positions:

```c
glPushMatrix();
  glTranslate(square 1 WS position);
  glRotate(square 1 orientation);
  Draw_square();
glPopMatrix();

glPushMatrix();
  glTranslate(square 2 WS position);
  glRotate(square 2 orientation)
  Draw_square();
glPopMatrix();
```
Lighting and Shading

Enabling lighting:  
`glEnable(GL_LIGHTING)();`

Select shading:  
`glShadeModel(GL_SMOOTH);`

Enable a light source:  
`glEnable(GL_LIGHT1);`

Specify parameters:  
`glLightf{iv}(GL_LIGHT0, param, value);`
Material Properties

To assign material properties:

\[ \text{glMaterial\{if\}v(face, param, value);} \]

\[ \text{GL_FRONT, GL_BACK, GL_FRONT_AND_BACK} \]

\[ \text{GL_AMBIENT, GL_DIFFUSE, GL_AMBIENT_AND_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS} \]
Backface Culling

Objects within the view-frustum may have polygons pointing away from the viewer
Not visible

Back-faces

The process is known as back-face culling

Facing away

Facing towards

Viewpoint
Backface Culling

To eliminate back-faces:

For each polygon in the scene {
    Take its normal vector
    Take the view direction vector
    Use the dot product to find the angle between normal and view direction
    If the angle is LESS than 90 degrees, then the polygon is *culled*
}
Visible Surface Determination

Painter's algorithm
Sort polygons relative to the viewpoint
Render those polygons that are nearer the viewpoint after those polygons that are further away from the viewpoint

Problems?
Visible Surface Determination

Painter's algorithm
Sort polygons relative to the viewpoint
Render those polygons that are nearer the viewpoint \textit{after} those polygons that are further away from the viewpoint

Problems?
Depth Buffer

Image-space visibility algorithm
Buffer is 2D array, one element per pixel
Compute depth of each generated pixel
Overwrite depth buffer value if new value is nearer to camera than previous
Non-linear, Z-fighting
Note: BSP Trees

Used by Quake, Quake 2, etc
World-space visibility algorithm
Visibility calculations on a *per polygon* basis
Split polygons
Compare with Z-buffer algorithm
Texturing

Bitmap (2D image) applied to a triangle
Each triangle associated with an image
Each vertex associated with a texture coordinate \((u,v)\)
For each rendered pixel:
  – Lookup corresponding texture element (texel)
  – Interpolation
Lab Help Sessions

• Visualisation (VIC) Studio:
  Thurs 27\textsuperscript{th} April, 10-12
  Thurs 4\textsuperscript{th} May, 13-15
  Thurs 11\textsuperscript{th} May, 15-17

• Purposes:
  – Ask questions/get help if in process of completing a lab task
  – Obtain feedback if you have work-in-progress
    • Code / report
    • Documentation