DH2323 DGI16

INTRODUCTION TO
COMPUTER GRAPHICS AND
INTERACTION

RASTERISED RENDERING
Part II

Christopher Peters
HPCViz, KTH Royal Institute of Technology, Sweden

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

Based on DGI12 notes by Carl Henrik Ek
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
Geometry Transformations

Matrix multiplication
Translation, scaling, rotation, projection
Familiar?
Perspective Projection

From OpenGL Programming Guide

---

Christopher Peters

DH2323 Rasterised Rendering II

chpeters@kth.se
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
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
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

Model and View Transform → Lighting → 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
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

GL_POINTS  GL_LINES  GL_LINE_STRIP  GL_LINE_LOOP

Polygon

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);
glEnd();
```

`glBegin` defines the start of a new geometric primitive:
3D vertices are defined using `glVertex3f`
The colour is defined using `glColor`

- 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
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGBA | GLUT_DEPTH | GLUT_DOUBLE);
glutCreateWindow("RGSquare Application");
glutReshapeWindow(400, 400);
```

Request for window size (not necessarily accepted)

RGB Colour, depth testing and double buffering

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

glutMainLoop(); \(\leftarrow\) Begin event loop
Some Glut Objects

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

void glutWireTorus(0.3, 1.5, 20, 20);
void glutSolidTorus(0.3, 1.5, 20, 20);
void glutSolidTeapot(1.0);
void glutSolidDodecahedron();
```
Matrices in OGL

To initialise a matrix in OpenGL, use

\texttt{glLoadIdentity();}

Clears the currently selected OpenGL matrix (projection, transform, texture) to the \textit{identity matrix}

\textit{Note: always be aware of what the matrix mode is activated!}

To select a matrix as the current matrix, use

\texttt{glMatrixMode(mode);}

\textit{Mode} can be one of:

\begin{itemize}
  \item GL\_PROJECTION
  \item GL\_MODELVIEW
  \item GL\_TEXTURE
\end{itemize}
Perspective Projection

\[ \text{glFrustum}(x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}, z_{\text{min}}, z_{\text{max}}); \]

- All points on the line defined by
  - \((x_{\text{min}}, y_{\text{min}}, -z_{\text{min}})\) and COP are mapped onto the lower left point on the viewport
  - \((x_{\text{max}}, y_{\text{max}}, -z_{\text{min}})\) and COP are mapped onto the upper right point on the viewport

Easier:
- \text{gluPerspective}(\text{fov, aspect, near, 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

\[
gluLookAt(\text{eye pos } x, \text{ eye pos } y, \text{ eye pos } z, \text{ look at } x, \text{ look at } y, \text{ look at } z, \text{ up } x, \text{ up } y, \text{ up } z);
\]

- Position camera at \text{eye pos } (x,y,z)
- Make camera viewing direction \text{look at } (x,y,z)
- Set camera up vector to \text{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 \textit{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{glRotateref(angle, vx, vy, vz);}\texttt{\textbackslash n}
Rotates around the axis \((vx, vy, vz)\) by \textit{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:

```c
glEnable(GL_LIGHTING);
```

Select shading:

```c
glShadeModel(GL_SMOOTH);
```

Enable a light source:

```c
glEnable(GL_LIGHT1);
```

Specify parameters:

```c
GLfloat light0_ambient[4] = { 0.2f, 0.2f, 0.2f, 1.0f; };
```

```c
GLfloat light0_diffuse[4] = { 0.5f, 0.5f, 0.5f, 1.0f; };
```

```c
GLfloat light0_specular[4] = { 1.0f, 1.0f, 1.0f, 1.0f; };
```

```c
GLfloat light0_position[4] = { 1.0f, 1.0f, 1.0f, 0.0f; };
```

```c
GLfloat light0_intensities[4] = { 1.0f, 1.0f, 1.0f, 1.0f; };
```

```c
glLightf(GL_LIGHT0, GL_AMBIENT, light0_ambient);
```

```c
glLightf(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
```

```c
glLightf(GL_LIGHT0, GL_SPECULAR, light0_specular);
```

```c
glLightf(GL_LIGHT0, GL_POSITION, light0_position);
```

```c
glLightf(GL_LIGHT0, GL_SPOT_Exponent, 20.0f);
```

```c
glLightf(GL_LIGHT0, GL_SPOT_FalloffConstant, 0.0f);
```

```c
glLightf(GL_LIGHT0, GL_SPOT_FalloffOuterCos, 0.0f);
```

```c
glLightf(GL_LIGHT0, GL_SPOT_FalloffInnerCos, 0.0f);
```

```c
glLightf(GL_LIGHT0, GL_SHADOW_CUTOFF, 0.0f);
```

```c
glLightf(GL_LIGHT0, GL_SHADOW_ATTENUATION_EXP, 0.0f);
```

```c
glLightf(GL_LIGHT0, GL_SHADOW_ATTENUATION_C0, 0.0f);
```

```c
glLightf(GL_LIGHT0, GL_SHADOW_ATTENUATION_C1, 0.0f);
```

```c
glLightf(GL_LIGHT0, GL_SHADOW_ATTENUATION_C2, 0.0f);
```

```c
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.0f);
```

```c
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.0f);
```

```c
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.0f);
```

```c
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, light0_direction);
```

```c
GLfloat light0_ambient[4] = { 0.2f, 0.2f, 0.2f, 1.0f; };
```

```c
GLfloat light0_diffuse[4] = { 0.5f, 0.5f, 0.5f, 1.0f; };
```

```c
GLfloat light0_specular[4] = { 1.0f, 1.0f, 1.0f, 1.0f; };
```

```c
GLfloat light0_position[4] = { 1.0f, 1.0f, 1.0f, 0.0f; };
```

```c
GLfloat light0_intensities[4] = { 1.0f, 1.0f, 1.0f, 1.0f; };
```
Material Properties

To assign material properties:

```
glMaterial{if}v(face, param, value);
```

- GL_FRONT
- GL_BACK
- GL_FRONT_AND_BACK

- GL_AMBIENT
- GL_DIFFUSE
- GL_AMBIENT_AND_DIFFUSE
- GL_SPECULAR
- GL_EMISSION
- GL_SHININESS
Conclusion

Fixed function pipeline is deprecated
  – Older OpenGL versions
  – Useful to gain understanding of basics

Compare to modern *programmable pipeline*
  – Vertex, pixel, etc shaders
  – Far more powerful
Links

http://nehe.gamedev.net
Legacy OGL tutorials (still great for learning)

http://www.khronos.org/webgl/
http://www.chromeexperiments.com/webgl/
WebGL, a graphics library for web browsers
Next lecture

- You should be working on Lab 3
- Next lab help session:
  13:00-15:00, Visualisation Studio,
  Thursday 5\textsuperscript{th} May
- Animation and image based rendering
- Next week (9\textsuperscript{th} May)
- 10:00 – 13:00