Chapter 2

Basics of Computer Graphics:
Rendering Pipeline/OpenGL

Course Info/Policies (boring stuff):
http://www.ugrad.cs.ubc.ca/~cs314
Grading

- Programming Assignments: 40%
  - Weekly Mini Home Quizes: 3%
  - Participation 2%
    - Classroom
    - Review question composition
  - Two Midterms: 25%
    - 12% +13%
  - Final Exam: 30%

---

Grading

- Programming Assignments: 40%
  - 2D Game: Intro to OpenGL (6%) - out now
  - 3D Transformations - modeling/animation (11%)
  - Rendering pipeline (11%)
  - Ray tracing (12%)
Grading

- Participation (2%)
  - Classroom: *Clicker* responses + classroom involvement
  - Post two weekly review questions
    - Based on material covered each week
    - Submit via DB (private, rev# tag)
      - till Mon 9AM
    - Include: question, multiple choice answers, explanation

Grading

- Mini Home Quizes: (3%)
  - Online (connect.ubc.ca) quiz each week (from week 2)
  - Released by Tue AM, Due Friday 9AM
  - Multiple choice questions
    - Student/instructor composed
    - If your question selected - *double your quiz grade !!!*
    - If two selected triple..
Important Dates

- Assignment 1 due: Sep 21
- Assignment 2 due: Oct 12
- Assignment 3 due: Nov 2
- Assignment 4 due: Nov 30
- Midterm 1: Oct 19
- Midterm 2: Nov 9

Course Organization

- Programming assignments:
  - C++, Windows or Linux
  - **Tested on department Linux machines**
  - OpenGL graphics library / GLUT for user interface

- **Face to face grading in lab**
  - Opportunity to show all the “cool” extra stuff
  - Test that you do know what every piece of your code does

- Hall of fame – coolest projects from 2002 on
Late/Missing Work

- Programming Assignments:
  - 3 grace days **TOTAL**
    - for unforeseen circumstances
    - strong recommendation: don't use early in term
    - handing in late uses up automatically unless you tell us
- Home Quizzes/Review Question Sets
  - Can miss *two* of each
- Exception: severe illness/crisis, as per UBC rules
  - MUST
    - Get approval from me ASAP (in person or email)
    - Turn in proper documentation

Literature (optional)

- Fundamentals of Computer Graphics
  - *Third edition (second is OK too – but note syllabus changes)*
  - Peter Shirley, A.K. Peters
- OpenGL Programming Guide
  - J. Neider, T. Davis and W. Mason, Addison-Wesley
Learning OpenGL

- This is a graphics course using OpenGL
  - not a course ON OpenGL
- Upper-level class: learning APIs mostly on your own
  - only minimal lecture coverage
    - basics, some of the tricky bits
  - OpenGL Red Book
  - many tutorial sites on the web

Plagiarism and Cheating

- Short Summary: Don’t cheat
  - Home quizzes and programming assignments are individual work
  - Can discuss ideas (including on DB), browse Web
  - But cannot copy code or answers/questions
    - If you REALLY think using a source is OK cite it
- Must be able to explain algorithms during face-to-face demo
  - or no credit for that assignment, possible prosecution
Basics of Computer Graphics: Rendering Pipeline

Goal:
- Transform (3D) computer models into images
- Photo-realistic (or not)

Interactive rendering:
- Fast, but until recently low quality
- Roughly follows a fixed patterns of operations
  - Rendering Pipeline

Offline rendering:
- Ray-tracing
- Global illumination
Rendering Tasks (no particular order)

- Project 3D geometry onto image plane
  - Geometric transformations
- Determine which primitives/parts of primitives are visible
  - Hidden surface removal
- Determine which pixels geometric primitive covers
  - Scan conversion
- Compute color of every visible surface point
  - Lighting, shading, texture mapping

The Rendering Pipeline

Geometric Content → Model/View Transform. → Lighting → Perspective Transform. → Clipping → Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer
Rendering Pipeline

- Abstract model of
  - sequence of operations to transform geometric model into digital image
  - graphics hardware workflow
- Underlying API (application programming interface) model for programming graphics hardware
  - OpenGL
  - Direct 3D

- Actual implementations vary

Advantages of pipeline structure?

- Logical separation of different components, modularity
- Easy to parallelize:
  - Earlier stages can already work on new data while later stages still work with previous data
  - Similar to pipelining in modern CPUs
  - But much more aggressive parallelization possible (special purpose hardware!)
  - Important for hardware implementations!
- Only local knowledge of the scene is necessary
Disadvantages?

- Limited flexibility
- Some algorithms would require different ordering of pipeline stages
  - Hard to achieve while still preserving compatibility
- Only local knowledge of scene is available
  - Shadows
  - Global illumination

(Tentative) Lecture Syllabus

- Introduction + Rendering Pipeline (week 1/2)
- Transformations (week 2/3)
- Scan Conversion (week 4/5)
- Clipping (week 5)
- Hidden Surface Removal (week 6/7)
- Review & Midterm (week 7)
  - Midterm: Oct 19
- Lighting Models (week 8)
- Texture mapping (week 9/10)
- Review & Midterm (week 10)
  - Midterm: Nov 9
- Ray Tracing (week 11)
- Shadows (week 11/12)
- Modeling (content creation) (week 12/13)
- Review (last lecture)
Rendering Pipeline Implementation: OpenGL/GLut

OpenGL

- API for graphics hardware
  - Started in 1989 by Kurt Akeley
  - Designed to exploit graphics hardware
  - Implemented on many different platforms

- Pipeline processing
  - Event driven
  - Communication via state setting
GLUT: OpenGL Utility Toolkit

- Event driven !!!

```c
int main(int argc, char **argv)
{
    // Initialize GLUT and open a window.
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
    glutInitWindowSize(800, 600);
    glutCreateWindow(argv[0]);

    // Register a bunch of callbacks for GLUT events.
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);

    // Pass control to GLUT.
    glutMainLoop();

    return 0;
}
```

Event-Driven Programming

- Main loop not under your control
  - vs. procedural
- Control flow through event callbacks
  - redraw the window now
  - key was pressed
  - mouse moved
- Callback functions called from main loop when events occur
  - mouse/keyboard, redrawing...

Copyright  A. Sheffer, 2012, UBC
Graphics State (global variables)

- Set state once, remains until overwritten
  - `glColor3f(1.0, 1.0, 0.0) → set color to yellow`
  - `glSetClearColor(0.0, 0.0, 0.2) → dark blue bg`
  - `glEnable(LIGHT0) → turn on light`
  - `glEnable(GL_DEPTH_TEST) → hidden surf.`

OpenGL/GLUT Example

```c
void display(void) {// Called when need to redraw screen.
    // Clear the buffer we will draw into.
    glClearColor(0, 0, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT);

    // Initialize the modelview matrix.
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Draw STUFF

    // Make the buffer we just drew into visible.
    glutSwapBuffers();
}
```
int main(int argc, char *argv[]) {
    ....
    // Schedule the first animation callback ASAP.
    glutTimerFunc(0, animate, 0);
    // Pass control to GLUT.
    glutMainLoop();
    return 0;
}

void animate(int last_frame = 0) {
    // Do stuff
    // Schedule the next frame.
    int current_time = glutGet(GLUT_ELAPSED_TIME);
    int next_frame = last_frame + 1000 / 30;
    glutTimerFunc(MAX(0, next_frame - current_time),
                   animate, current_time);
}

// you supply these kind of functions
void reshape(int w, int h);
void keyboard(unsigned char key, int x, int y);
void mouse(int but, int state, int x, int y);

    // register them with glut
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMouseFunc(mouse);
GLUT and GLU primitives

- `gluSphere(...)`
- `gluCylinder(...)`
- `glutSolidSphere(GLdouble radius, GLint slices, GLint stacks)`
- `glutWireSphere(...)`
- `glutSolidCube(GLdouble size)`
- `glutWireCube(...)`
- `glutSolidTorus(...)`
- `glutWireTorus(...)`
- `glutSolidTeapot(...)`
- `glutWireTeapot(...)`

**Note:**
- Have limited set of parameters
- Control via global transformations (see a1 template)
- **Need to save/ restore setting**

Example (from a1):

```cpp
void Pad::draw() {
    glColor3f(1, 1, 1);
    glPushMatrix();
    glTranslatef(x_, y_, 0);
    glScalef(width_, height_, 1);
    glNormal3f(0, 0, 1);
    glBegin(GL_QUADS);
    glVertex3f(-0.5, -0.5, 0);
    glVertex3f(-0.5, 0.5, 0);
    glVertex3f(0.5, -0.5, 0);
    glVertex3f(0.5, 0.5, 0);
    glEnd();
    glPopMatrix();
}  
```

→ Save previous state

→ Restore previous state
GLUT and GLU primitives

- Basic Transformations:

```c
// Different basic transformations
glTranslatef(...);
glRotatef(...);
glScalef(...);
```

Your tasks for the weekend

- Piazza Discussion Group:
  - Register
  - Post review questions by Mon 9AM
    - Use private option, rev1 tag

- Assignment 1
  - Test programming environment on lab computers/Set laptop environment (optional)
  - Should have all the necessary background after this class
Your tasks for the weekend

- Sign and Submit Plagiarism Form

- Optional reading (Shirley: Introduction to CG)
  - Math refresher: Chapters 2, 4
    - *Lots of math coming in the next few weeks*
  - Background on graphics: Chapter 1

Assignment 1

- Experience OpenGL & GLUT

- Have FUN


- Deadline: Sep 21
Rendering Pipeline in (More) Detail

The Rendering Pipeline

- Geometric Content
- Model/View Transform.
- Lighting
- Perspective Transform.
- Clipping
- Scan Conversion
- Texturing
- Depth Test
- Blending
- Frame-buffer

Copyright  A. Sheffer, 2012, UBC
3D Content

- Needs to represent models for
  - Shapes (objects)
  - Relations between different shapes
  - Object materials
  - Light sources
  - Camera

Shapes

- Different philosophies:
  - Volumetric
    - Boolean algebra with volumetric primitives
      - Spheres, cones, cylinders, tori, ...
  - Boundary representation
    - Single basic primitive
      - Triangles or triangle meshes, points, lines
    - Higher order surface primitives with adjustable parameters
      - E.g. “all polynomials of degree 2”
      - Splines, NURBS (details in CPSC 424)
      - Implicits
Curves/Surfaces

- Mathematical representations:
  - Explicit functions
  - Parametric functions
  - Implicit functions

Explicit Functions

- Curves:
  - $y$ is a function of $x$: $y := \sin(x)$
  - Only works in 2D

- Surfaces:
  - $z$ is a function of $x$ and $y$: $z := \sin(x) + \cos(y)$
  - Cannot define arbitrary shapes in 3D
Parametric Functions

- Curves:
  - 2D: \( x \) and \( y \) are functions of a parameter value \( t \)
  - 3D: \( x \), \( y \), and \( z \) are functions of a parameter value \( t \)

\[
C(t) := \begin{pmatrix} \cos(t) \\ \sin(t) \\ t \end{pmatrix}
\]

- Surfaces:
  - Surface \( S \) is defined as a function of parameter values \( s, t \)
  - Names of parameters can be different to match intuition:

\[
S(\phi, \theta) := \begin{pmatrix} \cos(\phi)\cos(\theta) \\ \sin(\phi)\cos(\theta) \\ \sin(\theta) \end{pmatrix}
\]
Shapes

- Implicit Surfaces:
  - Surface defined by zero set (roots) of function
  - E.g:

$$S(x, y, z) : x^2 + y^2 + z^2 - 1 = 0$$

Shapes

- Triangles and Triangle Meshes:
  - How to define a triangle?
Open GL: (Some) Shape Primitives

- **glPointSize(float size);**
- **glLineWidth(float width);**
- **glColor3f(float r, float g, float b);**

```
   GL_TRIANGLES
   glBegin(GL_TRIANGLES);
   glVertex3f(0.5f, 0.5f, 0.0f);
   glVertex3f(-0.5f, 0.0f, 0.0f);
   glVertex3f(-0.5f, -0.5f, 0.0f);
   glEnd();
```

---

**Open GL – Shape Primitives**

- How to interpret geometry
  - **glBegin(<mode of geometric primitives>);**
  - mode = GL_TRIANGLES, GL_POLYGON, etc.

- Feed vertices
  - `glVertex3f(-1.0, 0.0, -1.0)`
  - `glVertex3f(1.0, 0.0, -1.0)`
  - `glVertex3f(0.0, 1.0, -1.0)`

- Done
  - `glEnd()`
The Rendering Pipeline

- **Geometric Content**
  - Model/View Transform.
  - Lighting
  - Perspective Transform.
  - Clipping

- **Scan Conversion**
  - Texturing
  - Depth Test
  - Blending
  - Frame-buffer

- **Rasterization**
  - Fragment Processing

**Modeling and Viewing Transformations**

- Placing objects - Modeling transformations
  - Map points from object coordinate system to world coordinate system

- Placing camera - Viewing transformation
  - Map points from world coordinate system to camera (or eye) coordinate system

Copyright A. Sheffer, 2012, UBC
Viewing Transformation: Camera Placement

Modeling & Viewing Transformations

- Types of transformations:
  - Rotations, scaling, shearing
  - Translations
  - Other transformations (not handled by rendering pipeline):
    - Freeform deformation
Modeling & Viewing Transformation

- Linear transformations
  - Rotations, scaling, shearing
  - Can be expressed as 3x3 matrix
  - E.g. scaling (non uniform):
    \[
    \begin{pmatrix}
    x' \\
y' \\
z'
    \end{pmatrix} = \begin{pmatrix}
    2 & 0 & 0 \\
    0 & 3 & 0 \\
    0 & 0 & 1
    \end{pmatrix} \cdot \begin{pmatrix}
    x \\
y \\
z
    \end{pmatrix}
    \]

- Affine transformations
  - Linear transformations + translations
  - Can be expressed as 3x3 matrix + 3 vector
  - E.g. scale+ translation:
    \[
    \begin{pmatrix}
    x' \\
y' \\
z'
    \end{pmatrix} = \begin{pmatrix}
    2 & 0 & 0 \\
    0 & 3 & 0 \\
    0 & 0 & 1
    \end{pmatrix} \cdot \begin{pmatrix}
    x \\
y \\
z
    \end{pmatrix} + \begin{pmatrix}
    t_x \\
t_y \\
t_z
    \end{pmatrix}
    \]

- Another representation: 4x4 homogeneous matrix
The Rendering Pipeline

Geometric Content → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Geometry Processing

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer

Rasterization → Fragment Processing

Lighting

Copyright A. Sheffer, 2012, UBC
Complex Lighting and Shading

The Rendering Pipeline

Geometric Content → Model/View Transform → Lighting → Perspective Transform → Clipping

Geometry Processing

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer

Rasterization

Fragment Processing
Perspective Transformation

- Purpose:
  - Project 3D geometry to 2D image plane
  - Simulates a camera

- Camera model:
  - Pinhole camera (single view point)
  - Other, more complex camera models also exist in computer graphics, but are less common
    - Thin lens cameras
    - Full simulation of lens geometry

Perspective Projection
Perspective Transformation

- In computer graphics:
  - Image plane conceptually in front of center of projection
  - Perspective transformations - subset of projective transformations
  - Linear & affine transformations also belong to this class
  - All projective transformations can be expressed as 4x4 matrix operations

The Rendering Pipeline

<table>
<thead>
<tr>
<th>Geometric Content</th>
<th>Model/View Transform.</th>
<th>Lighting</th>
<th>Perspective Transform.</th>
<th>Clipping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan Conversion</td>
<td>Texturing</td>
<td>Depth Test</td>
<td>Blending</td>
<td>Frame-buffer</td>
</tr>
</tbody>
</table>

Rasterization | Fragment Processing
Clipping

- Removing invisible geometry
  - Geometry outside viewing frustum
  - Plus too far or too near one

The Rendering Pipeline
Scan Conversion/Rasterization

- Convert continuous 2D geometry to discrete
- Raster display - discrete grid of elements

Terminology

- **Pixel**: basic element on device

- **Resolution**: number of rows & columns in device
  - Measured in
    - Absolute values (1K x 1K)
    - Density values (300 dots per inch)

- **Screen Space**: Discrete 2D Cartesian coordinate system of the screen pixels
Scan Conversion

- Problem:
  - Line is infinitely thin, but image has finite resolution
  - Results in steps rather than a smooth line
    - Jaggies
    - Aliasing
  - One of the fundamental problems in computer graphics
Scan Conversion

- Color interpolation
  - Linearly interpolate per-pixel color from vertex color values
  - Treat every channel of RGB color separately
Color interpolation

Example:

The Rendering Pipeline

Geometric Content → Model/View Transform. → Lighting → Perspective Transform. → Clipping

Geometry Processing

Scan Conversion → Texturing → Depth Test → Blending → Frame-buffer

Rasterization

Fragment Processing
Texturing

\[(s_0, t_0)\]

\[(s_1, t_1)\]

\[(s_2, t_2)\]

\[s\]

\[t\]
Texture Mapping

Displacement Mapping
Texturing

- Issues:
  - Computing 3D/2D map (low distortion)
  - How to map pixel from texture (texels) to screen pixels
    - Texture can appear widely distorted in rendering
    - Magnification / minification of textures
  - Filtering of textures
  - Preventing aliasing (anti-aliasing)

The Rendering Pipeline
Without Hidden Line Removal

Hidden Line Removal
Hidden Surface Removal

- Remove invisible geometry
  - Parts that are hidden behind other geometry
- Possible Implementations:
  - Per-fragment decision
    - Depth buffer
  - Object space decision
    - Clipping polygons against each other
    - Sorting polygons by distance from camera
Depth Test / Hidden Surface Removal

The Rendering Pipeline

- Geometric Content
- Model/View Transform
- Lighting
- Perspective Transform
- Clipping

- Scan Conversion
- Texturing
- Depth Test
- Blending
- Frame-buffer

Copyright A. Sheffer, 2012, UBC
Blending

- Blending:
  - Final image: write fragments to pixels
  - Draw from farthest to nearest
  - No blending – replace previous color
  - Blending: combine new & old values with some arithmetic operations
- Frame Buffer: video memory on graphics board that holds resulting image & used to display it

Not Handled: Reflection/Shadows