16 - Additional Topics

Game-Loop Approaches
- Tightly-Coupled
- Fixed frame rate
- Variable-timestep update()
- Gameloop-managed update timing
- Fixed logic rate
- Renderer frame interpolation
- Multi-threading

Review: Game Structure

One iteration = 1 Frame

Tightly-Coupled Approach

Difficulties:
- Different machine speeds produce different game flow effects
- Changes to game logic (e.g., AI) can slow down presentation (frame rate)

Problem source:
- The two steps have different inherent frequencies
  - logicUpdate() frequency should be fixed
  - render() frequency should optimize hardware use

Conclusion:
- Need to somehow “decouple” the steps

Fixed frame rate
Constant frame – and update – rate (one update per frame)

Variable-timestep updating

update() is responsible for accounting for time variation
Gameloop-Managed Timed update

Game-loop accounts for time variation

- desiredUpdateTime = \(\frac{1}{\text{desiredUPS}}\times1000000000\);
- lastUpdateTime = System.nanoTime();
- while (!gameOver)
  - curTime = System.nanoTime();
  - timeSinceLastUpdate = curTime - lastUpdateTime;
  - if (timeSinceLastUpdate > desiredUpdateTime)
    - handleInput();
    - update();
    - lastUpdateTime = System.nanoTime();
  - render();

Interpolated frames

- render() assists in accounting for time variation (e.g. by smoothing animations)

- desiredUpdateTime = \(\frac{1}{\text{desiredUPS}}\times1000000000\);
- startTime = System.nanoTime();
- while (!gameOver)
  - currentTime = System.nanoTime();
  - timeSinceLastUpdate = currentTime - startTime;
  - if (timeSinceLastUpdate > desiredUpdateTime)
    - handleInput();
    - update();
    - startTime += updatePeriod;
  - percentWithinUpdatePeriod = \(\min\{1.0, \frac{(currentTime - startTime)}{\text{desiredUpdateTime}}\}\);
  - render(percentWithinUpdatePeriod);

Multi-threaded Approach

Separate threads for Update() and Render()

- Logic Update loop:
  - Runs at controlled (platform-independent) speed
  - Performs world state update, AI calculations, etc.

- Presentation loop:
  - Runs “as fast as possible”
  - Draws visible world
  - Makes maximum use of hardware capabilities

Game Engine Timing Support

- AbstractGame
  - BaseGame
    - # mainLoop()

- VariableFrameRateGame
  - final mainLoop(){
  - final getFramesPerSec();
  - final setMaxFrameRate()

- FixedLogicRateGame
  - final mainLoop(){
  - final getUpdatesPerSec()

- UserGame
  - update(); render()
Values in normal maps are usually stored as offsets from vertical relative to the plane tangent to the surface, with the Z (or B) coordinate set to 1.0. This is why normal maps appear “bluish.” The surface tangent plane is defined by mutually-perpendicular tangent and bi-tangent vectors (both perpendicular to the normal).

Moon example:

Sphere with moon texture map:

Sphere textured with corresponding normal map:

Normal map lighting effects on moon:

Combining texture map and lighting with normal map -- effect on moon:

Note the changes in specular highlights at the indicated locations.
**Height Mapping**

Using a texture image to perturb the vertex locations.

*Use a grayscale image, where:*

- **White** == “high”
- **Black** == “low”

**Example:**

- height map
- standard texture

Result when applied to a 100x100 rectangular grid

**Texture-Mapped Fonts**

Create a texture map image containing font characters:

- “texture map” each desired character to a single polygon
- Build output image strings from texture-mapped polygons
Texture-Mapped Fonts (cont.)

Drawbacks:
- Requires an external application to create the font texture
- Scaling can affect font appearance
- Difficult to implement proportionally-spaced fonts
- Uses up a texture unit

see the Wikibook "Modern OpenGL Tutorial Text Rendering 01"

Billboards

Basic approach:
- Replace a complex 3D model with a 2D image
- Maintain image orientation toward camera

Works well when
- Model is sufficiently “far away”
- Model tends to have “axial symmetry”

Basic steps:
- Create flat object with 2D texture-mapped image
- Translate object to tree location in world
- Rotate object so that it points to the camera

Atmospheric Effects - Fog

A useful effect: blend pixel color with another color (e.g. gray) based on distance from eye.

“Fog” is not just for simulating fog, but also for enhancing the sense of 3D depth for the viewer.

There are simple models, as well as sophisticated models that include light scattering effects.

a more comprehensive model:

Extinction – rate of decay from image color to black
Inscattering – rate of change from image color to fog color

\[ f_e = e^{-xz} \]
\[ f_i = e^{-zs} \]

\( z = \text{distance from eye to object}, \)
\( x = \text{extinction coefficient}, \)
\( s = \text{inscattering coefficient} \)
The importance of shadows

???

???

Shadow Mapping

• (“pass 1”) Render the scene from the light's position. The depth buffer then contains, for each pixel, the distance between the light and the nearest object to it.
• Copy the depth buffer to a separate “shadow buffer”.
• (“pass 2”) Render the scene normally. For each pixel:
  ➢ look up the corresponding position in the shadow buffer
  ➢ If the distance to the point being rendered is greater than the value retrieved from the shadow buffer
    • the object being drawn at this pixel is further from the light than the object nearest the light, and
    • therefore this pixel is in shadow
    • make the pixel darker
  ➢ else draw it normally

Soft Shadows

There are techniques for simulating the “soft shadows” that occur in nature (not covered here)

Atmospheric Effects – Environment Mapping

• OpenGL cubemap = a single texture object with six 2D faces
• Texture coords (s,t,r) = vector from the cube center

using OpenGL cubemaps makes “environment mapping” easier
**Environment Mapping**

- Useful technique for rendering “mirror” objects
- Create texture cube map describing the “environment”
- Shade object points by following “reflection” of eye vector (to surface normal) into the map

**3D Textures**

- 2D Texture Images are “painted on” and sometimes they look like it
- Desired: make it look like an object is “made out of” some material

**3D Texture Example: Wood-grain**

Wood grain is caused by *tree rings*

- Needed: a function \( f(x, y, z) \) mapping an object location into the “wood space” – i.e. obtaining the “wood color”

**Wood-grain example (cont.)**

Computing cylindrical location for \((x, y, z)\) object point

\[
R = \sqrt{x^2 + z^2} \\
\theta = \tan^{-1}\left(\frac{y}{\sqrt{x^2+z^2}}\right) \\
H = y
\]
Perlin Noise

Ken Perlin (NYU), 1985
- "Classic Noise" (1985)
  - Gradient noise using cubic Hermite interpolation
- "Improved Noise" (2001)
  - Gradient noise using quintic interpolation and precomputed gradients
- "Simplex Noise" (2005)
  - Hardware implementation

Computer Graphics
V19 No.3, July 1989
AMPAS Oscar, 1997

"Randomness" is important in many 3D textures:
- Natural materials (wood, marble, granite …)
- Man-made materials (stucco, asphalt, cement …)
- Natural phenomena (clouds, fire, smoke, wind effects …)
- Model imperfections (rust, dirt, smudges, dents …)
- Pattern and motion imperfections (bumps, wobbles, jitters …)

Image source: ACM SIGGRAPH Education Slide Set, R. Wolfe, DePaul Univ.

Example – “Jade”

Atmospheric Effects - Clouds

Clouds are complex:
- scatter and reflect light in diverse ways
- drift and morph over time

There are simple models, as well as sophisticated models that include light scattering effects.
Simple models often use 3D textures and Perlin noise.

Atmospheric Effects - Clouds

Mixing 3D noise at different levels of precision:
Atmospheric Effects - Clouds

3D texture allows efficient "morphing" by slowly changing the Z parameter when accessing texture map:

```
3D texture allows efficient "morphing" by slowly changing the Z parameter when accessing texture map:
```

“Immediate-Mode” Graphics

CPU transfers RAM data over main bus

- Time consuming
- Bus Contention

```
#include(GL_TRIANGLES);
setColor3f(1.0, 0.0, 0.0);
vertex3f(0.0, 0.0, 0.0);
setColor3f(0.0, 1.0, 0.0);
vertex3f(1.0, 0.0, 0.0);
setColor3f(0.0, 0.0, 1.0);
vertex3f(1.0, 1.0, 0.0);
setColor3f(1.0, 1.0, 0.0);
vertex3f();
```

The PROGRAMMABLE Graphics Pipeline

Programmable Shaders:

- Vertex shader
- Tessellation Control
- Tessellation Evaluation
- Geometry Shader
- Fragment Shader

Hardware (“Shader”) Programming

High-level languages:
- HLSL (“High-Level Shading Language”)
  - Proprietary (Microsoft)
  - Powerful
  - Specific to DirectX
- Cg (“C for graphics”)
  - Proprietary (nVidia)
  - Supports both DirectX and OpenGL APIs (more complex)
- GLSL (“OpenGL Shading Language”)
  - Open standard
  - Compiles to all common vendor chips
  - Can run "on top of" DirectX, or directly on hardware

Vertex Arrays

```
Vertex array pointer

x y z x y z x y z x y z ...
```

```
Vertex Geometry Array

Texture

Coord

pointer

5 1 5 1 5 1 5 1 5 1 ...
```

Steps to use Vertex Arrays:
- Enable use of each array type (geometry, tex, etc.)
- Point to each array to be used
- Load each array with data (geometry, tex, etc.)
- Specify what to draw, and from where, in the arrays
Indexed Vertex Arrays

Vertex Buffer Objects (VBOs)

- Map an “application buffer” directly to high-speed video memory (such as DRAM)
- Buffers can include “vertex arrays”
- Methods like `glDrawArrays()` operate on video memory data

see CSc-155…