CIS 441/541: Introduction to Computer Graphics
Lecture 14: Final project, buffers

Nov. 21st, 2014
Hank Childs, University of Oregon
Back from NOLA...
Announcements

- Midterms graded and returned Nov 26th
- Project proposal due Nov 26th
  - Ideally you will do it earlier and get going earlier
- Everything back to normal on Nov 21st (NOW)
- 2B due Nov 24th
Outline

- Final projects
- Review
- Buffers
Outline

- Final projects
- Review
- Buffers
Jordan Weiler project
Adam Zucker project
Zhuojun (Morgan) Zhang
(it walks)
Rajat
Guidelines

- Do something that excites you
- Should be a significant project
- Should involve graphics in a serious way
  - Including implementation of graphics code
    - Example: Use program to generate geometry, but ALSO add graphics codes to see that geometry in a cool way
- I don’t care what technology you use: VTK, GLUT, GL, etc.
Ideas

- Screen saver
- Modeling (+ graphics)
- Video Game
- Visualization
- Advanced animation
- Advanced graphics techniques
- really anything...
Deliverable:
- Email me a written proposal
- 1-2 paragraphs
- In word/PDF, not an email
- not graded

Goal: make sure we don’t find out we disagree on the day of the final

Deadline: Nov 26
- Will turn them around as they come in
  - (so you can hand it in earlier if you want)

Office Hours will change to support your projects
Outline

- Final projects
- Review
- Buffers
What is going on here?

**Details:**
- Diffuse = 0.8, ambient = 0.2
- Default GL normal is (0,0,1)

```c++
virtual void RenderPiece(vtkRenderer *ren, vtkActor *act)
{
    RemoveVTKOpenGLStateSideEffects();
    SetupLight();
    glBegin(GL_TRIANGLES);
    glVertex3f(-10, -10, -10);
    glVertex3f(10, -10, 10);
    glVertex3f(10, 10, 10);
    glEnd();
}
```
But wait…

If you have an open surface, then there is a “back face”. The back face has the opposite normal.

How can we deal with this case?

Idea #1: encode all triangles twice, with different normals
Idea #2: modify diffuse lighting model

This is called two-sided lighting

Diffuse light = abs(L · N)
Reminder: open surface, closed surface

- **Closed surface:**
  - you could drop it in water and it would float (i.e., no way for water to get inside)
  - no way to see the inside

- **Open surface:**
  - water can get to any part of the surface
  - you can see the inside
Front face / back face

- Front face: the face that is “facing outward”
- Back face: the face that is “facing inward”
- These distinctions are meaningful for closed surfaces
- They are not meaningful for open surfaces
  - This is why we did “two-sided lighting”
What is going on here?

**Answer:**
OpenGL is giving us diffuse+ambient on the left, but only ambient on the right (rather, diffuse == 0).

Why? ... it is interpreting the left side as a “front face” and the right side as a “back face”.

(if we spun the camera, the left would be gray, and the right would be white)
The front face and back face is determined by convention.

Convention #1:

V1, V2, and V3 are arranged counter-clockwise around the front-face.
The front face and back face is determined by convention

Convention #1:

Which vertex is specified first doesn’t matter, as long as they are arranged counter-clockwise around the front-face.
The front face and back face is determined by convention.

Convention #2:

V1, V2, and V3 are arranged clockwise around the front-face.
The front face and back face is determined by convention.

- **Convention #1:**
  - Front-face: V1
  - Back-face: V2

- **Convention #2:**
  - Front-face: V2
  - Back-face: V3
Name

glFrontFace — define front- and back-facing polygons

C Specification

```c
void glFrontFace(GLenum mode);
```

Python Specification

```python
glFrontFace(mode) → None
```

Parameters

- `mode`

  Specifies the orientation of front-facing polygons. `GL_CW` and `GL_CCW` are accepted. The initial value is `GL_CCW`.

Description

In a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible. Eliminating these invisible polygons has the obvious benefit of speeding up the rendering of the image. To enable and disable elimination of back-facing polygons, call `glEnable` and `glDisable` with argument `GL_CULL_FACE`.

The projection of a polygon to window coordinates is said to have clockwise winding if an imaginary object following the path from its first vertex, its second vertex, and so on, to its last vertex, and finally back to its first vertex, moves in a clockwise direction about the interior of the polygon. The polygon's winding is said to be counterclockwise if the imaginary object following the same path moves in a counterclockwise direction about the interior of the polygon. `glFrontFace` specifies whether polygons with clockwise winding in window coordinates, or counterclockwise winding in window coordinates, are taken to be front-facing. Passing `GL_CCW` to `mode` selects counterclockwise polygons as front-facing; `GL_CW` selects clockwise polygons as front-facing. By default, counterclockwise polygons are taken to be front-facing.
Why front face / back face is important

- **Reason #1:**
  - Need to know this if you are doing lighting calculations

- **Reason #2:**
  - Culling
Culling

- Idea: some triangles can’t affect the picture, so don’t render them.
- Question: how can this happen?
- Answer #1: geometry outside [0->width, 0->height] in device space
- Answer #2: closed surface, and the back face is facing the camera
glCullFace — specify whether front- or back-facing facets can be culled

**C Specification**

```c
void glCullFace(GLenum mode);
```

**Parameters**

*mode*

Specifies whether front- or back-facing facets are candidates for culling. Symbolic constants `GL_FRONT`, `GL_BACK`, and `GL_FRONT_AND_BACK` are accepted. The initial value is `GL_BACK`.

**Description**

`glCullFace` specifies whether front- or back-facing facets are culled (as specified by *mode*) when facet culling is enabled. Facet culling is initially disabled. To enable and disable facet culling, call the `glEnable` and `glDisable` commands with the argument `GL_CULL_FACE`. Facets include triangles, quadrilaterals, polygons, and rectangles.

`glFrontFace` specifies which of the clockwise and counterclockwise facets are front-facing and back-facing. See `glFrontFace`. 
Something you should know about: Left- and right-handed coordinates

OpenGL: right-handed
DirectX: left-handed
Animation Software: Maya

Overview

Comprehensive 3D animation software

Autodesk® Maya® 3D animation software offers a comprehensive creative feature set for 3D computer animation, modeling, simulation, rendering, and compositing on a highly extensible production platform. Maya now has next-generation display technology, accelerated modeling workflows, and new tools for handling complex data.

Maya features

Next-gen viewport display and shading
Harness the power of Microsoft® DirectX® 11 software with the Maya DX11 Shader.
See all features

Accelerated modeling workflow
Increase productivity with accelerated polygon modeling workflows.

Scene Assembly tools for smarter data
Create large, complex worlds more easily with new Scene Assembly tools.

Get more in a suite

Reference
Scene Assembly
Asset
Select Set Members

Extend your animation workflow with Autodesk® Entertainment Creation Suite.
Explore Entertainment Creation Suite.

https://www.youtube.com/watch?v=kyTryl1jp30
https://www.youtube.com/watch?v=kyTryl1jp30
Let's do an example of CCW + right-handed coordinates
Final projects
Review
Buffers
Buffers

- OpenGL buffers:
  - Color buffer
  - Depth buffer
  - Stencil buffer
  - Accumulation buffer ← not always supported

- By the way, did you know that these calls toggle depth buffering?
  
  ```c
  glEnable(GL_DEPTH_BUFFER) /
  glDisable(GL_DEPTH_BUFFER)
  ```
A stencil buffer is an extra buffer found on modern graphics hardware. The buffer is per pixel, and works on integer values, usually with a depth of one byte per pixel. The depth buffer and stencil buffer often share the same area in the RAM of the graphics hardware.
A screenshot from a program which uses that stencil buffer to change the color of shapes as they pass over other shapes. The stencil buffer is filled with 1s wherever a white stripe is drawn and 0s elsewhere. Two versions of each oval, square, or triangle are then drawn, and the stencil buffer is used to decide per-pixel which object to draw. A black colored shape is drawn where the stencil buffer is 0, and a white shape is drawn where the buffer is 1.
Stencil buffer uses

- Many sophisticated uses.
- Traditional example:
  - Windshield for flight simulator
    - Define a region that should not be rendered, then GPU will only render in the remaining region.
Name

glClear — clear buffers to preset values

C Specification

void glClear(GLbitfield mask);

Parameters

mask

Bitwise OR of masks that indicate the buffers to be cleared. The three masks are GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, and GL_STENCIL_BUFFER_BIT.

Description

glClear sets the bitplane area of the window to values previously selected by glClearColor, glClearDepth, and glClearStencil. Multiple color buffers can be cleared simultaneously by selecting more than one buffer at a time using glDrawBuffer.

The pixel ownership test, the scissor test, dithering, and the buffer writemasks affect the operation of glClear. The scissor box bounds the cleared region. Alpha function, blend function, logical operation, stenciling, texture mapping, and depth-buffering are ignored by glClear.

glClear takes a single argument that is the bitwise OR of several values indicating which buffer is to be cleared.

The values are as follows:

GL_COLOR_BUFFER_BIT

Indicates the buffers currently enabled for color writing.

GL_DEPTH_BUFFER_BIT

Indicates the depth buffer.

Example:

glClear called between every render ... it is like your init code in project 1F.

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glReadPixels - read a block of pixels from the frame buffer

C SPECIFICATION

```c
void glReadPixels( GLint x,
                  GLint y,
                  GLsizei width,
                  GLsizei height,
                  GLenum format,
                  GLenum type,
                  GLvoid *pixels )
```

PARAMETERS

**x, y** Specify the window coordinates of the first pixel that is read from the frame buffer. This location is the lower left corner of a rectangular block of pixels.

**width, height**
Specify the dimensions of the pixel rectangle. **width** and **height** of one correspond to a single pixel.

**format**

**type**
Specifies the data type of the pixel data. Must be one of **GL_UNSIGNED_BYTE**, **GL_BYTE**, **GL_BITMAP**, **GL_UNSIGNED_SHORT**, **GL_SHORT**, **GL_UNSIGNED_INT**, **GL_INT**, **GL_FLOAT**, **GL_UNSIGNED_BYTE_3_3_2**, **GL_UNSIGNED_BYTE_2_3_3_REV**, **GL_UNSIGNED_SHORT_5_6_5_REV**, **GL_UNSIGNED_SHORT_4_4_4_4**, **GL_UNSIGNED_SHORT_4_4_4_4_REV**, **GL_UNSIGNED_SHORT_5_5_5_1**, **GL_UNSIGNED_SHORT_1_5_5_5_REV**.
NAME

**glDrawPixels** - write a block of pixels to the frame buffer

C SPECIFICATION

```c
void glDrawPixels( GLsizei width, 
    GLsizei height, 
    GLenum format, 
    GLenum type, 
    const GLvoid *pixels )
```

PARAMETERS

- **width**, **height** Specify the dimensions of the pixel rectangle to be written into the frame buffer.

- **format** Specifies the format of the pixel data. Symbolic constants GL_COLOR_INDEX, GL_STENCIL_INDEX, GL_DEPTH_COMPONENT, GL_RGB, GL_BGR, GL_RGBA, GL_BGRA, GL_RED, GL_GREEN, GL_BLUE, GL_ALPHA, GL_LUMINANCE, and GL_LUMINANCE_ALPHA are accepted.

- **type** Specifies the data type for **pixels**. Symbolic constants GL_UNSIGNED_BYTE, GL_BYTE, GL_BITMAP, GL_UNSIGNED_SHORT, GL_SHORT, GL_UNSIGNED_INT, GL_INT, GL_FLOAT, GL_UNSIGNED_BYTE_3_3_2, GL_UNSIGNED_BYTE_2_3_3_REV, GL_UNSIGNED_SHORT_5_6_5, GL_UNSIGNED_SHORT_5_6_5_REV, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_4_4_4_4_REV, GL_UNSIGNED_SHORT_5_5_5_1, GL_UNSIGNED_SHORT_1_5_5_5_REV, GL_UNSIGNED_INT_8_8_8_8, GL_UNSIGNED_INT_8_8_8_8_REV, GL_UNSIGNED_INT_10_10_10_2, and GL_UNSIGNED_INT_2_10_10_10_REV are accepted.

- **pixels** Specifies a pointer to the pixel data.
Accumulation buffer

- Buffer with 64 bits of resolution
- The color buffer can be accumulated into this buffer.
- After a while, the accumulation can be put into the color buffer.
NAME

glAccum - operate on the accumulation buffer

C SPECIFICATION

    void glAccum( GLenum op,
                  GLfloat value )

PARAMETERS

    op       Specifies the accumulation buffer operation. Symbolic constants GL_ACCUM, GL_LOAD, GL_ADD, GL_MULT, and GL_RETURN are accepted.

    value    Specifies a floating-point value used in the accumulation buffer operation. op determines how value is used.

DESCRIPTION

The accumulation buffer is an extended-range color buffer. Images are not rendered into it. Rather, images rendered into one of the color buffers are added to the contents of the accumulation buffer after rendering. Effects such as antialiased (of points, lines, and polygons), motion blur, and depth of field can be created by accumulating images generated with different transformation matrices.

Each pixel in the accumulation buffer consists of red, green, blue, and alpha values. The number of bits per component in the accumulation buffer depends on the implementation. You can examine this number by calling glGetIntegerv four times, with arguments GL_ACCUM_RED_BITS, GL_ACCUM_GREEN_BITS, GL_ACCUM_BLUE_BITS, and GL_ACCUM_ALPHA_BITS. Regardless of the number of bits per component, the range of values stored by each component is [-1, 1].

The accumulation buffer pixels are mapped one-to-one with frame buffer pixels.

glAccum operates on the accumulation buffer. The first argument, op, is a symbolic constant that selects an accumulation buffer operation. The second argument, value, is a floating-point value to be used in that operation. Five operations are specified: GL_ACCUM, GL_LOAD, GL_ADD, GL_MULT, and GL_RETURN.

All accumulation buffer operations are limited to the area of the current scissor box and applied identically to the red, green, blue, and alpha components of each pixel. If a glAccum operation results in a value outside the range [-1, 1], the contents of an accumulation buffer pixel component are undefined.

The operations are as follows:

GL_ACCUM       Obtains R, G, B, and A values from the buffer currently selected for reading (see glReadBuffer). Each component value is divided by (2^n)-1, where n is the number of bits allocated to each color component in the currently selected buffer. The result is a floating-point value in the range [0, 1], which is multiplied by value and added to the corresponding pixel component in the accumulation buffer, thereby updating the accumulation buffer.

GL_LOAD        Similar to GL_ACCUM, except that the current value in the accumulation buffer is not used in the calculation of the new value. That is, the R, G, B, and A values from the currently selected buffer are divided by (2^n)-1, multiplied by value, and then stored in the corresponding accumulation buffer cell, overwriting the current value.

GL_ADD          Adds value to each R, G, B, and A in the accumulation buffer.

GL_MULT         Multiplies each R, G, B, and A in the accumulation buffer by value and returns the scaled component to its corresponding accumulation buffer location.

GL_RETURN       Transfers accumulation buffer values to the color buffer or buffers currently selected for writing. Each R, G, B, and A component is multiplied by value, then multiplied by (2^n)-1, clamped to the range [0,(2^n)-1], and stored in the corresponding display buffer cell. The only fragment operations that are applied to this transfer are pixel ownership, scissor, dithering, and color writemasks.

To clear the accumulation buffer, call glClearAccum with R, G, B, and A values to set it to, then call glClear with the accumulation buffer enabled.
Accumulation buffer: motion blur
Jitter

- Jitter is a technique that deals with line aliasing that sometimes is performed using the accumulation buffer.

- What is line aliasing?
Line Aliasing

- Ideal raster line is one pixel wide
- All line segments, other than vertical and horizontal segments, partially cover pixels
- Simple algorithms color only whole pixels
- Lead to the “jaggies” or aliasing
- Similar issue for polygons
OpenGL Antialiasing

- Can enable separately for points, lines, or polygons

```c
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```
void display(void)
{
    GLint viewport[4];
    int jitter;

    glGetIntegerv (GL_VIEWPORT, viewport);

    glClear(GL_ACCUM_BUFFER_BIT);
    for (jitter = 0; jitter < ACSIZE; jitter++) {
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        accPerspective (50.0,
                        (GLdouble) viewport[2]/(GLdouble) viewport[3],
                        1.0, 15.0, j8[jitter].x, j8[jitter].y, 0.0, 0.0, 1.0);
        displayObjects ();
        glAccum(GL_ACCUM, 1.0/ACSIZE);
    }
    glAccum (GL_RETURN, 1.0);
    glFlush();
}
Filling a pool with a bucket

- Your goal:
  - Fill an empty swimming pool

- The situation:
  - Hose dumps 1 gallon / 20s
  - One bucket, which holds 1 gallon
  - 50ft from hose to pool, takes 10s to make a round trip
  - Pool holds 1000 gallons
  - Can’t spill water

- The question:
  - How long to fill the pool?
Filling a pool with a bucket

- Your goal:
  - Fill an empty swimming pool

- The situation:
  - Hose dumps 1 gallon / 20s
  - Two buckets, each holds 1 gallon
  - 50ft from hose to pool, takes 10s to make a round trip
  - Pool holds 1000 gallons
  - Can’t spill water

- The question:
  - How long to fill the pool?
Filling a pool with a bucket

- Your goal:
  - Fill an empty swimming pool

- The situation:
  - Hose dumps 1 gallon / 20s
  - Three buckets, each holds 1 gallon
  - 50ft from hose to pool, takes 10s to make a round trip
  - Pool holds 1000 gallons
  - Can’t spill water

- The question:
  - How long to fill the pool?
Relation to Computer Graphics

- Filling the bucket
  \[\text{\rightarrow}\]\n- the process of rasterizing triangles

- Walking to the pool and dumping the water
  \[\text{\leftrightarrow}\]\n- the process of taking the resulting image and putting it onto the screen

If you have one bucket, then you have to stop the hose while dumping it.
If you have one place to render triangles, then you have to stop rendering while displaying the current image.
Multiple buffering

- In computer science, multiple buffering is the use of more than one buffer to hold a block of data, so that a "reader" will see a complete (though perhaps old) version of the data, rather than a partially updated version of the data being created by a "writer".
A software implementation of double buffering has all drawing operations store their results in some region of system RAM

- this region is called a "back buffer"

When all drawing operations are considered complete, the whole region (or only the changed portion) is copied into the video RAM (the "front buffer")
Double buffering in CG

- Double buffering necessarily requires more memory and CPU time than single buffering because of:
  - the system memory allocated for the back buffer
  - the time for the copy operation
  - the time waiting for synchronization
In computer graphics, triple buffering is similar to double buffering but provides a speed improvement.

In double buffering the program must wait until the finished drawing is copied or swapped before starting the next drawing.

This waiting period could be several milliseconds during which neither buffer can be touched.
Triple buffering (!!)

- In triple buffering the program has two back buffers and can immediately start drawing in the one that is not involved in such copying.
- The third buffer, the front buffer, is read by the graphics card to display the image on the monitor.
- Once the monitor has been drawn, the front buffer is flipped with (or copied from) the back buffer holding the last complete screen.
- Since one of the back buffers is always complete, the graphics card never has to wait for the software to complete.
Quad buffering (!!!!!)

- The term "Quad buffering" is used in stereoscopic implementations, and means the use of double buffering for each of the left and right eye images, thus four buffers total.
- The command to swap or copy the buffer typically applies to both pairs at once.
  - If triple buffering was used then there would be six buffers. (!!!!!!!!!!)
NAME

glFlush - force execution of GL commands in finite time

C SPECIFICATION

void glFlush( void )

DESCRIPTION

Different GL implementations buffer commands in several different locations, including network buffers and the graphics accelerator itself. glFlush empties all of these buffers, causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine. Though this execution may not be completed in any particular time period, it does complete in finite time.

Because any GL program might be executed over a network, or on an accelerator that buffers commands, all programs should call glFlush whenever they count on having all of their previously issued commands completed. For example, call glFlush before waiting for user input that depends on the generated image.

NOTES

glFlush can return at any time. It does not wait until the execution of all previously issued GL commands is complete.

ERRORS

GL_INVALID_OPERATION is generated if glFlush is executed between the execution of glBegin and the corresponding execution of glEnd.

SEE ALSO

glFinish
What is multi-pass rendering?

- Multipass rendering refers to a set of techniques in 3D computer graphics.

- In multipass rendering, a single 3D object (or scene) is rendered multiple times.
  - Each time the object is drawn, an additional aspect of object's appearance is calculated and combined with the previous results.
  - The process of rendering the object is called a pass.