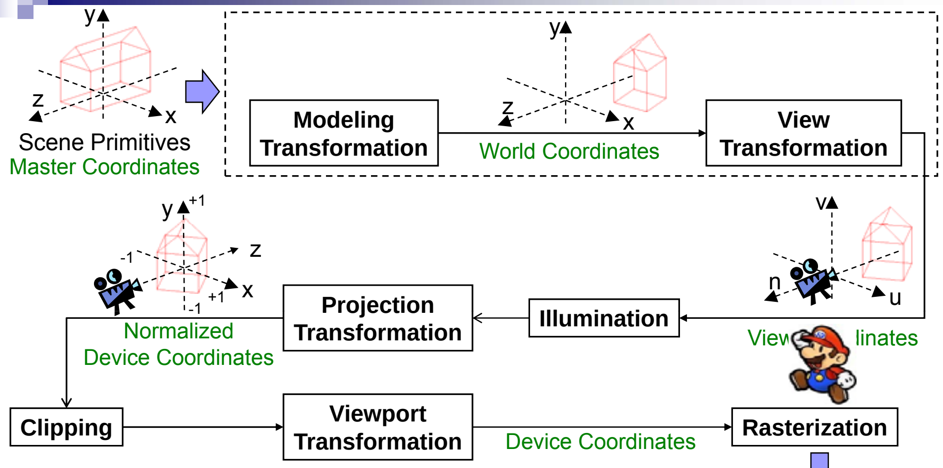
Computer Graphics: Rasterization I

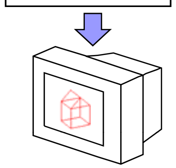
Part 2 – Lecture 12

Today's Outline

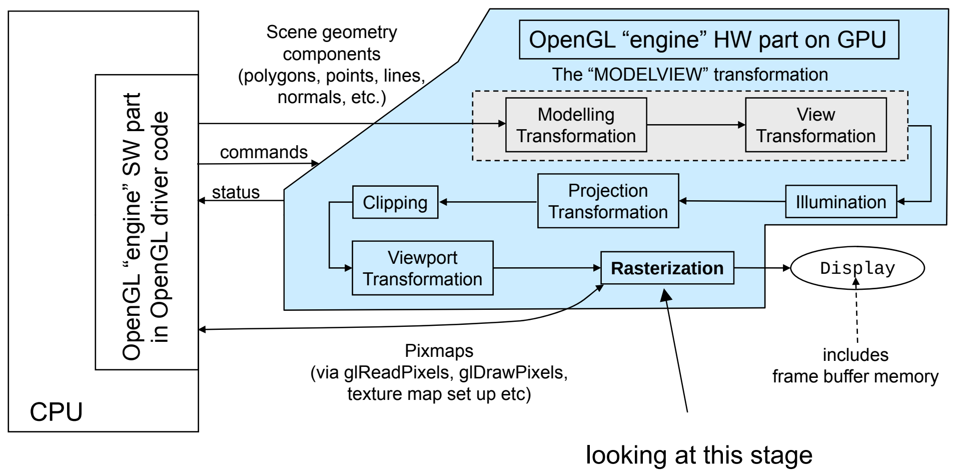
- Rasterization in OpenGL
- Pixmaps and Blending
- OpenGL Display Lists



RASTERIZATION IN OPENGL



Rasterization Stage of Rendering Pipeline



Rasterization Stage

- **Input**: scene component geometry from viewport transformation, vertex and normal coordinates (3D, floating point)
- Rasterization = converting floating point numbers that define primitives into "rasters", i.e. pixels in frame buffer memory
- **Output**: coordinates and colors of pixels that comprise primitives' shapes in the frame buffer array

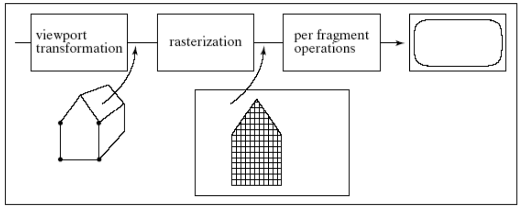


FIGURE 10.1 The rasterization step in the graphics pipeline.

Rasterization Operations

- **Point rasterization**: convert (x,y,z) vertex to "disc" (filled circle) of pixels, dependent upon `glPointSize`
- **Line rasterization**: convert 2 (x,y,z) vertices to sequence of pixels, dependent upon `glLineWidth` (and other functions such as `glLineStipple`)
- **Polygon rasterization**: convert n (x,y,z) vertices to 2D region of pixels, dependent upon many functions, e.g. `glPolygonMode`

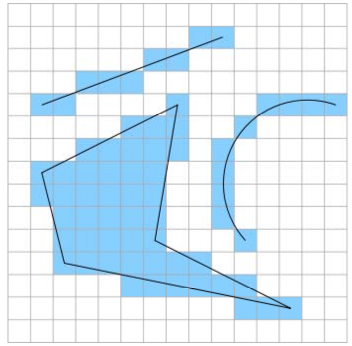
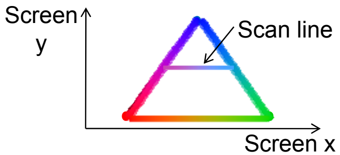


Image thanks to Wojciech Muła

Other Rasterization Operations

- **Shading**: flat or smooth (Gouraud) Color interpolation along scanline → can be reduced to simple additions, e.g. $red_{pixel\ (i+1)} = red_{pixel\ i} + delta_{red}$

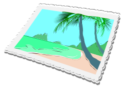


- **Depth buffer (z-buffer) calculations**:
  1. Compute each pixel's z value (as an integer) → can be reduced to simple additions, e.g. $z_{pixel\ (i+1)} = z_{pixel\ i} + delta_z$
  2. If computed pixel z value < current z-buffer depth value
     1. Replace z-buffer value at that pixel location with computed z
     2. Replace color buffer values at that pixel location with computed color (from shading algorithm)
- **Other per-pixel operations**: texture map interpolation, anti-aliasing and other blending ops, pixmap ops (text, overlay, compositing, etc.)

PIXMAPS AND BLENDING

Pixmaps in OpenGL

- Arrays of pixels, usually used to store an image, e.g.
  - Pixels saved from the frame buffer (rendering window content, "screen dump")
  - Imported image, e.g. from a file
- Examples of tasks that use pixmaps:
  - Read all or part of an image rendered by an OpenGL program and store it in a file
  - Write images onto a screen object




  - Write bitmap of text (font defined by 1 bitmap per character)
  - Write menu items, button labels, icons, etc. onto a GUI

  - Copy a 2D "sprite" from one region of screen to another (also for scrolling)


Different Types of Pixmaps

Pixel formats: what components to store per pixel

- Color buffer values: `GL_RGBA`, `GL_RGB`, `GL_RED`, `GL_GREEN`, `GL_BLUE`, `GL_ALPHA`
- Color index values: `GL_COLOR_INDEX`
- Intensity values: `GL_LUMINANCE`, `GL_LUMINANCE_ALPHA`
- Depth buffer (z-buffer) values: `GL_DEPTH_COMPONENT`
- Stencil buffer values: `GL_STENCIL_INDEX`

Data types: how to store each component

- 1 bit: `GL_BITMAP`
- 1 byte: `GL_UNSIGNED_BYTE`, `GL_BYTE`
- integer: `GL_UNSIGNED_SHORT`, `GL_SHORT`, `GL_UNSIGNED_INT`, `GL_INT`
- float: `GL_FLOAT`

Reading Pixmap & Setting Raster Position

- Reading pixels into an array:

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

  - *x, y, width, height* defines the region of pixels
  - *pixels* is the pointer to the array (needs to be big enough)
- Setting the current raster position:

```
void glRasterPos3f(GLfloat x, GLfloat y, GLfloat z)
```

  - Sets the current raster position in 4D world space (x, y, z, w)
  - Note: raster position is transformed by current modelview and projection matrices
  - Can also set raster position directly in window coordinates:

```
void glWindowPos2i(GLint x, GLint y)
```

Writing & Copying a Pixmap

- Writing pixmap from array to current raster position:

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

  - *width, height* defines size of pixel region
  - *pixels* is the pointer to array with pixmap to be drawn

- Copying pixmap from frame buffer to current raster position:

```
void glCopyPixels(GLint x, GLint y,
                  GLsizei width, GLsizei height,
                  GLenum type)
```

  - *type* specifies one of 3 possible buffer types that can be copied: `GL_COLOR`, `GL_DEPTH` or `GL_STENCIL`

Automatic Operations on Pixels

Scale and bias pixel values while transferred from/to frame buffer:

```
void glPixelTransferf(GLenum pname, GLfloat param)
```

- destination value = source value * scale + bias
- *pname* selects operation: `GL_RED_SCALE`, `GL_ALPHA_SCALE`, `GL_DEPTH_SCALE`, `GL_RED_BIAS`, …
- *param* sets scaling or bias value

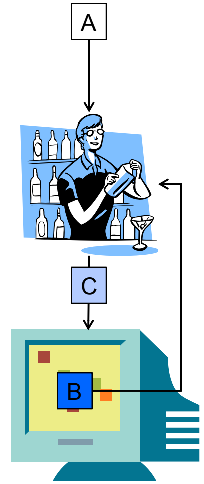
Zoom pixmaps that are written/copied to frame buffer:

```
void glPixelZoom(GLfloat xfactor, GLfloat yfactor)
```

- Magnifies or reduces written/copied pixmap by replicating/ommitting pixels
- Can also be used for mirroring with negative zoom factor

Blending Pixel Values

- Blending in OpenGL uses read-modify-write cycle: When new pixel A is written, it is combined with the pixel B that is already there, resulting in pixel C
- Blending is done with $C_{i,j} = a\ A_{i,j}$ PIXEL_OP $b\ B_{i,j}$
  - PIXEL_OP may be any arithmetic or logical function
  - Multiplication coefficients *a* and *b* can be set with pixel alpha values (opacity)
- Examples:
  - $C_{i,j} = ½\ A_{i,j} + ½\ B_{i,j}$ (averaging)
  - $C_{i,j} = A_{i,j} - B_{i,j}$ (differencing)
  - $C_{i,j} = t\ A_{i,j} + (1 – t)\ B_{i,j}$ (linear interpolation, "fade" , "dissolve")



Choosing the Blending Coefficients

```
void glBlendFunc(GLenum sfactor, GLenum dfactor)
```

- $C_{i,j}$ = `sfactor` * $A_{i,j}$ PIXEL_OP `dfactor` * $B_{i,j}$
- *sfactor* may be: `GL_ZERO, GL_ONE, GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA GL_SRC_ALPHA_SATURATE`
  Default: `GL_ONE`
- *dfactor* may be: `GL_ZERO, GL_ONE, GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA`
  Default: `GL_ZERO`
- Enable/disable blending with `glEnable/glDisable` and `GL_BLEND`

Choosing the Blending Function

```
void glBlendEquation(GLenum mode)
```

- Use it to choose blending functions other than addition
- *mode* is one of `GL_FUNC_ADD, GL_FUNC_SUBSTRACT, GL_FUNC_REVERSE_SUBSTRACT, GL_MIN, GL_MAX, GL_LOGIC_OP`

```
void glLogicOp(GLenum opcode)
```

- Use it to select sepcific `GL_LOGIC_OP` blending function

opcode	**value**	*opcode*	**value**
GL_CLEAR	0	GL_AND	A & B
GL_SET	1	GL_OR	A | B
GL_COPY	A	GL_XOR	A ^ B

- `glEnable/glDisable` using `GL_COLOR_LOGIC_OP`

OPENGL DISPLAY LISTS

Immediate Mode Execution

OpenGL engine (HW and SW driver) processes commands from scratch every time the `display()` function is called:

1. Function calls (e.g. glBegin/glEnd, glVertex, glNormal) must be translated into driver and hardware commands ("assembly language" for the GPU)
2. Commands and data values must be copied from CPU memory into the GPU's local memory (on graphics card)

→ Efficient if commands or data change frequently (e.g. vertex values are recomputed in each call to `display()`)

→ But with constant commands and data this is very inefficient!

Retained Mode Execution

- If commands and data are constant, prepare them in advance (like a compilation step):
  1. Request OpenGL to construct a **display list** (with an integer id)
  2. Use the same function calls (including C++ statements such as loops, etc.) and data values

  → Commands are translated into GPU code and copied with data from CPU memory to GPU memory and stored in the GPU
- To render the display list, only one command is sent to OpenGL (copied from CPU to GPU): **`glCallList`**`(idNumber);`

Immediate mode execution is similar to interpreting source code

Compiling a display list and retained mode execution is like compiling and executing source code

OpenGL Display Lists

1. Get *range* unused *`listIds`* for your display lists (first id is returned): `GLuint` **`glGenLists`**`(GLSsizei` *`range`*`)`
2. Start list definition with call to
   `void` **`glNewList`**`(GLuint` *`listId`*`, GLenum` *`mode`*`)`
   (*mode* is either `GL_COMPILE` or `GL_COMPILE_AND_EXECUTE`)
3. Follow this with all code (OpenGL calls and C++) for rendering the objects to be included in this list
   - Not all commands are stored (e.g. no state queries)
   - May include execution of other display lists
   - May not call `glNewList`
4. End list definition with call to `void` **`glEndList`**`()`

→ Execute the display list with
`void` **`glCallList`**`(Gluint` *`listId`*`)`

Display Lists: Pros and Cons

Advantages of using a display list (→ retained mode)

- Speed up (compared to immediate mode) can be significant
- Modular reuse of commands and data
  - Set state appropriately before calling display list (e.g. transformations, colors, …)
  - Call other display lists from within a display list

Disadvantages of using a display list (→ immediate mode)

- If data or commands change frequently, using a display list may be slower (list cannot be changed, has to be compiled again)
- Display lists do not allow parameter passing (except setting of appropriate state before calling the list)

SUMMARY

Summary

1. **Rasterization**: converting floating point primitives into pixels + shading + depth testing + blending
2. **Pixmap operations**: read, write, copy
3. **Blending** with existing pixels when writing new pixels: weighted sum, difference, min, max, logic operations, …
4. **Display lists** can be used to compile a list of commands and data for faster execution

References:

- Rasterization: Hill, Chapter 9.1
- Pixmaps and Blending: Hill, Chapter 9.2 – 9.3
- OpenGL API Reference: http://www.cs.auckland.ac.nz/compsci372s1c/resources/manpagesOpenGL

Quiz

1. What is done during the rasterization stage?
2. What can we do with pixmaps?
3. What is blending and how can we blend pixels in OpenGL?
4. What are display lists and why are they useful?