

Today we're going to discuss graphics software systems in general. Thurs. we'll start talking specifically about OpenGL/GLUT

What do you think I mean by a graphics system?

What are some graphics systems/packages/libraries you're familiar with?

OpenGL

GL

GKS

QuickDraw

SRGP

PHIGS

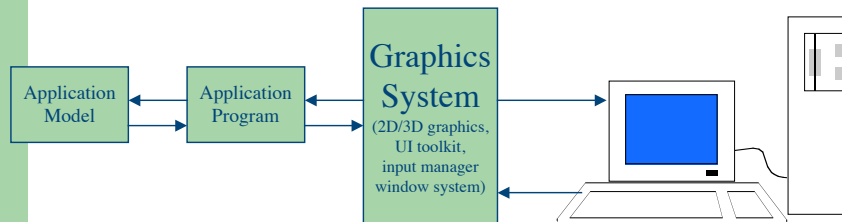
Core

Postscript

X11

What is a graphics package?

- software
 - that takes user input and passes it to applications
 - that displays graphical output for applications



2

(C) Doug Bowman, Virginia Tech, 2007

In the most general terms, a graphics system is just the graphics part of the software that sits in between the hardware and application programs

As you can see, this covers a lot of ground

Application

- Model (world):
 - the database & objects to be displayed
 - may be geometry/attributes
 - may be abstract data (e.g. fractal description)
- Program:
 - responsible for mapping the model to primitives supported by graphics package
 - responsible for mapping user input to model changes

3

(C) Doug Bowman, Virginia Tech, 2007

Let's go through each of the parts of the diagram (we know what the hardware is - just input and output devices)

The application model (also sometimes called the world) is the program's representation of what's going to be displayed. It may be a description of geometry and attributes (e.g. "a blue sphere centered at 0,0,0 with radius 5.0") or it may be abstract (e.g. a mathematical formula for determining a fractal)

This distinction between the model/world and the screen will be important later on in the class as well.

The application program's job (among other things) is to take the model and transform it into something that can be understood by the graphics package

It also takes the user input (passed through the graphics system) and maps it to changes in the model

Graphics system components

- Set of primitives
- Primitive attributes
- Graphics output
- Input handling
- Window management

As we saw in the diagram, the graphics system actually contains several components

2D Primitive possibilities

- geometrical objects (point, line, circle, polygon, ...)
- mathematical curves
- text primitives
- fill patterns
- bitmapped images/textures

Primitives are just objects (like graphical data types) that are supported by a graphics system.

Here are some possible 2D primitives (not an exhaustive list).

3D primitive possibilities

- geometrical objects (line, polygon, polyhedron, sphere, ...)
- mathematical surfaces
- light sources
- camera/eye points
- hierarchy placeholders
- object boundaries

Here are some possible primitives for 3D graphics

Primitive attributes

- color
- thickness
- position
- orientation
- transparency
- behavior

The GS also allows the application to set attributes or properties for graphics primitives - here are some possibilities.

Graphics output

- *rendering*: process of mapping graphics commands (sets of primitives/attributes) to pixel values to be placed in frame buffer
- Mapping from application/world space into screen space
- providing a *view* of the application model

Once the application has specified primitives and attributes, the GS is responsible for realizing those primitives in terms of output on the screen.

Again, you can think of this as a mapping or transformation - from the more abstract primitive descriptions to actual pixel values. Called rendering of primitives

The idea of different spaces also comes up here - we're mapping from model/world space into screen space (different coordinate systems!)

A final way to think of this job is providing the user a certain view into the model (a window on the internal world of the application) - that is often the point of computer graphics - visualization of something that otherwise is only present in bits.

Input handling

- receive input from physical devices
- map this input to logical devices for applications
- apps register interest in events or devices
- event-based programming

```
render_from_database();  
while(1){  
    wait for input  
    switch(input){  
        case 1: call_routine1();  
        case 2: call_routine2();  
        ...  
    }  
    render_from_database();  
}
```

The GS is also responsible for handling input from the user, since it sits between the application and the devices.

Again, think of this as a mapping/transformation - we're taking the physical input and mapping it to logical devices (remember that?) so that applications can make sense of it.

How does the GS know what to do with an input? Applications usually have to explicitly express interest in something (e.g. a right mouse button click) - then the GS will pass it on to that app (assuming that other conditions are met - such as window was in focus, etc.)

This leads to a model of programming called event-driven programming (anyone have any experience with this?)

-different from normal sequential program

-structure: init, run event loop, quit

-the GS basically renders the scene, waits for an event, passes it on, and re-renders the scene

Any problems with this setup (e.g. for animation/VR - can't wait on events)

Styles of graphics programming

- Event-based
 - Uses an event loop
 - Events (user input) placed into queues by I/O subsystem
 - When an event occurs, a callback function runs to respond to the event
 - Normally the scene is only redrawn after an event (application model changed)
- Simulation-based
 - Uses a simulation loop
 - Also uses event queues and callback functions
 - BUT, other processing (simulation) is done continuously even without user input
 - The scene is redrawn continuously as quickly as possible

Window management

- manage screen space
- mediate between application programs
- provide logical output “canvases”
- each app. believes it has an entire “screen” with its own coordinate system

Window managers/window systems are usually separate entities from the graphics package.

Their job is to manage the available screen space and mediate this space between multiple applications

This is where logical output devices come in - each application only sees its canvas(es) and doesn't need to worry about everyone else.

The window manager takes care of saving portions of windows that get covered up, dividing the space among windows, deciding the size and position of windows, etc.

Goals of graphics packages

- Abstraction; Device-independence
 - logical input devices
 - logical output devices (!)
 - provide abstraction from hardware for app.
 - produce application portability
- Appropriate primitive/attribute types

We've talked about the structure of graphics packages, now how do we know if we've got a good one?

The first goal is abstraction (what is abstraction?)

-mainly in this case we're abstracting away from actual devices by providing logical input and output devices

-this produces applications that are portable

-the actual GS code must be implemented for each platform

-early GS's were platform-dependent and very low-level

-also, the GS abstracts away from pixels and allows app. programs to think in terms of complete primitives

The second goal is to have appropriate primitive/attribute types

-what does this mean?

-appropriate for the most common applications?

-appropriate level of abstraction (low-level: draw_pixel, high-level: draw_fractal)

-what are the tradeoffs between high and low-level primitives?

OpenGL

- mid-level, device-independent, portable graphics subroutine package
- 2D/3D graphics, lower-level primitives (polygons)
- does not include low-level I/O management
- basis for higher-level libraries/toolkits

we're going to be programming in OpenGL

its main primitive is the polygon (2D/3D) although it includes a lot more

it is not concerned with low-level input handling, window management, etc.

however, it is also not always the level immediately below applications - there are a lot of higher-level libraries/toolkits built on top of OpenGL

ex: SVE

OpenGL is state-based

- Commands:
 - depend on current state
 - modify the current state
- Example: `glVertex3i(x, y, z)` depends on
 - current primitive mode
 - current draw color
 - current material and lighting state
 - ...
- Example: `glColor3f(r, g, b)` modifies the current drawing color

OpenGL geometric primitives

- Points
- Lines
- Line strips
- Line loops
- Polygons
- Triangles
- Triangle strips
- Triangle fans
- Quads
- Quad strips

OpenGL graphics pipeline

