CS 378: Computer Game Technology

http://www.cs.utexas.edu/~fussell/courses/cs378/

Spring 2013

Instructor and TAs

- Instructor: Don Fussell
  - fussell@cs.utexas.edu
  - Office: ACES 2.120 and then GDC 5.510
  - Office Hours: TTh 2:00-3:00 or by appointment
- TA
  - Randy Smith
  - agrippa@cs.utexas.edu
  - Office: TBD
  - Office Hours: TBD

Communications

- We're using Piazza for class communication this semester
  - Piazza supports group based question and answer sessions among the entire class and with instructors
  - To enroll, go to http://piazza.com/utexas/spring2013/cs378gt
  - Most of you will probably have been pre-enrolled automatically
  - Once enrolled, your course page is at http://piazza.com/utexas/spring2013/cs378gt/home
  - You must enroll, class announcements will be done via Piazza.
  - You should enroll, this is a highly recommended system for enhancing the discussion of issues and questions that arise during class.
  - Please let me or Randy know asap if you have trouble enrolling/using this
- Grade feedback will be done via Blackboard, https://courses.utexas.edu/
- I won't be using Blackboard for other things, so please look at the webpage and Piazza for all else.

The Computer Game Industry

- Hardware makers produce gaming hardware
  - e.g. Sony, Nintendo, Microsoft, …
- Game Developers create games
  - e.g. Electronic Arts (EA), Epic, ID, thousands of others
- Publishers publish games
  - e.g. Sony, Nintendo, EA, …
- Similar to books or movies
  - One group creates it, another distributes it, and another supplies the underlying infrastructure

Game Development Team

- *Game Designers* decide on the format and behavior of the game
- *Artists* design models, textures, animations and otherwise are responsible for the look of the game
- *Level Designers* create the spaces in which the game takes place
- *Audio Designers* are responsible for all the sounds used in the game
- *Programmers* write code, to put it all together, and tools, to make everyone else's job simpler
- And others: Production, management, marketing, quality assurance

Course Outline

- Real-time graphics
  - Rendering pipeline review
  - Lighting, shaders, effects
  - Object, level, and character modeling
  - Scene graphs and 3d engines
  - Data management
- Game physics
  - Rigid body physics
  - Collision detection
  - Particle systems, fluid dynamics
  - Character animation
  - Other techniques
- Game AI
  - Character behavior
  - Path planning
- Networking
  - Managing delays and bandwidth
  - Security and cheating
- Other aspects of game engines
  - UI, sound, etc.

Take note

- This list and order is tentative, it will surely evolve as the course goes on.
- This is a game technology course, not a game design course. We're leaving a lot of stuff out.
- New developers at game companies aren't going to be engine programmers out of the box.
- What are they then?
  - General purpose problem solvers
  - Tool maintainers and possibly tool developers and enhancers
  - Experts at understanding complex software tools they didn't write
  - Productive in difficult situations

Books

- Recommended "textbook": "Game Engine Architecture" Jason Gregory
  - Good exposition of many issues in the technology and design of game engines
  - Not required (yet), but very useful
  - Not nearly as expensive as a textbook
- Other useful books:
  - "Game Programming Gems 1-8"
  - "3D Game Engine Design" David Eberly, lots of equations, less exposition, good math background, computer graphics with game slant
  - There are zillions of game books, I'm not familiar with most of them
  - This stuff is linked from the class webpage
- Website: www.gamasutra.com
  - Game developer technical and trade news
  - Other specific web sites

What you should already know

- 3D graphics concepts and programming
  - "Standard" lighting and shading
  - Modeling techniques
  - Vectors, matrices, geometric reasoning
  - OpenGL will be the graphics API discussed in lectures, but we won't be working primarily at that level
- C++ programming
- Familiarity with at least one user interface toolkit e.g. FLTK, Glut, Qt, …
  - We may not use any of these, but if you've used one, you'll have an idea what's happening
- Scripting language experience a plus, e.g. Lua, Python
- How to deal when you're required to use something new, but we don't teach it to you

Grading and workload

- Projects, not tests (they don't have them in game companies)
  - 1 – 2 major projects, broken into smaller pieces, each graded separately
  - Work in small groups of 3 people depending on task and class size
  - I will assign groups, you don't get to pick them yourselves.
  - I expect groups to be different at various points in the semester, so you won't always be working with the same set of people.
  - Not all projects will be just programming.

Working in Groups

- Working in groups is not easy, and it is an acquired skill
- It might be the most important thing you learn in here
- For some information on group functioning, read
  - http://www-honors.ucdavis.edu/vohs/index.html
- We will assign them, like in industry, you don't get to choose who you work with
- There will be some group evaluation exercises through the semester

Projects

- The initial project is a ball game
- Features:
  - Linux, not windows, is the “release” platform
  - You can develop on your own machines, integrate on windows
  - Use source code control
  - Limited game design, it’s a pinball game, but there are lots of variants
  - Limited 3d game, sound, limited UI
  - 2d physics
  - No animation, not too much object design (ball, bumpers, table, flippers, etc.)
  - Groups
  - Timeline
  - Grading
  - Tools – a big part of this is familiarizing yourself with toolchain
- Depending on how this goes, there may be a followon project
  - Add AI, more controls
  - More game design

Timeline

- Something due roughly every 2 weeks throughout the semester, weekly milestones
- First stage: Basic functionality
- Subsequent stages:
  - 3d camera positioning
  - Physics
  - Sound effects
  - Gui controls
  - Networking
  - AI
- Refinement of earlier stages is allowed

Grading

- Groups will be graded as one, but there may be adjustments for individuals
- Each group will set goals for the stage
  - Advice will be given on reasonable goals
  - Goals will be discussed and recorded near the start of each stage
  - Goals can be modified in the face of problems
- You will be graded based on how well you achieve your goals, with a degree of difficulty factor
- Each stage will involve turning in some kind of artifact, possibly involving a demo

Tools

- Your project must run and be turned in on the 64bit Linux machines in the ENS basement
- We plan to use Ogre3d as the 3d engine.
- We'll recommend sound packages, UI packages, etc. but you can choose your own within limits. These will be part of your proposal stages, and choosing and evaluating such tools is very important.
- We'll use bullet for physics
- You can develop on your own machines, but integrating code and running demos has to be done on the lab machines.
- You may spend a surprising amount of time getting your tools installed, working, and playing together. That's part of the problem solving part. And the last part is, effectively, building a game engine
- Source code control systems, essential for team projects

Tools for content creation

- Models and art are the biggest expense in real games
  - We try to limit these requirements as much as possible
  - You can use the open source modeler Blender in the lab or on your own machines. It has a steep learning curve, like other powerful modelers.
  - Building models by hand might be the most efficient option
  - You are free to use any available tools, provided you acknowledge it
  - You can use things that run on your own machines and maybe not on Linux if you like. The requirement is that the resulting assets be integrated on the Linux environment on which your game is delivered
- Textures should be a lesser problem
- Be prepared to write small tools if you think it will make your project easier
- Also, be prepared to write format converters if you have a good tool that produces output that your game engine can't input. This is a big deal in the real world as well as in class.

Interactive programming

- Games are user-controlled interactive programs
  - To be effective, they must be responsive to user input in real time, i.e. control must be direct and immediate.
  - Any good interactive program should provide constant, up-to-date feedback about its state. This is especially true of a game.
  - Users must know and understand what is happening at all times.
  - Users must get immediate feedback so they know their input was received.
- Effective interaction is a key component of immersion
  - A game is an alternate reality.
  - Not only must it be visually convincing, it must convince the user that they are in that world by reacting to user behavior as the real world would.

Structure of interactive programs

- Event-driven programming
  - Everything happens in response to events.
  - Events can occur asynchronously wrt the execution of the program reacting to the event.
  - Events can come from users (user control actions) or from system components.
  - Events can cause signals or messages to be generated to be sent to a system component. So events, signals, messages are in some sense equivalent.

User generated events

- When you press a button on a joystick, that's an event.
- The joystick hardware sends a signal to the computer called an interrupt.
- The OS responds to the interrupt by converting it to an item in an "event queue" for the windowing system.
- These events can be kept in priority order, temporal order, etc.
- UI toolkits have API elements for checking and responding to events.
- There are basically two ways for your software to know an event has happened, ask or be told.

Polling vs. waiting

- Most window systems provide a call to check if an event is pending that returns immediately, whether or not there is an event (nonblocking).
- So, if there's not one, what do you do?
- Loop to keep checking? Go off and do something else for awhile?
- Alternatively, there is also a blocking event function that waits (blocks) until an event has arrived, and only then returns.
- So, what happens while your program waits? Does any work get done? Does the screen freeze up?

System generated events

- Timer events
  - Application calls a function requesting an event at a future time, e.g. next time a frame should be drawn.
  - The system provides an event at the requested time.
  - Application checks for and responds to the event, e.g. by drawing the next frame.

Work procedures

- Some systems provide a function to be called when no event is pending.
- A blocking event check call executes this function if no event is found, rather than just waiting.
- Problems?
  - Are you sure this would ever be called?
  - How long does the function take to execute?

Callbacks

- Why not just tell the system in advance what to do when a particular type of event arrives, and let it happen automatically rather than you writing code to check events and execute something when they arrive?
- Most GUI systems operate this way, through a mechanism of callbacks.
- Your application makes a call to the GUI to tell it what function should be executed when a particular kind of event arrives.
  - E.g. when a timer event arrives, the system will call a draw function. You make a call ahead of time to give the system a pointer to the draw function you want it to use.
  - E.g. when the left mouse button is clicked, the system calls the mouse event function. You provide a pointer to the function you want called to handle mouse events.

Event response classes

- Two fundamental kinds of event responses:
- Mode change events
  - Cause the system to shift to a different mode of operation
- Task events
  - Cause the system to perform a specific task within a mode of operation
- Game software structure reflects this
  - Menu system is separate from game runtime, for instance

Real-time event loops

- Games and similar interactive systems look like an big infinite loop:

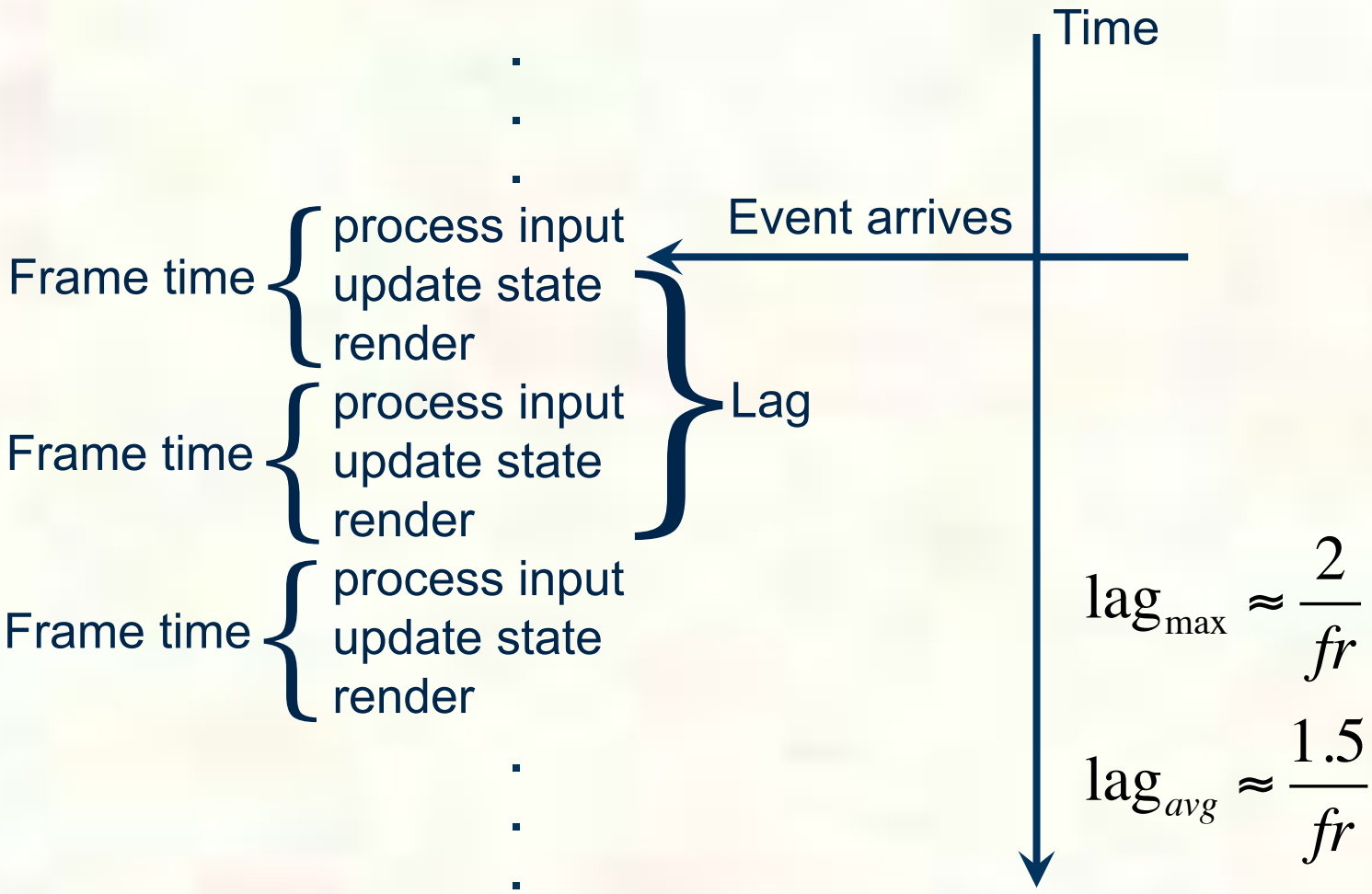
```
while (1) {
        process events
        update state
        render
}
```

- The number of times this loop executes per second is the frame rate (since each render operation creates a new frame).
- Measured in frames per second (fps).

Latency and lag

- Latency is the time it takes from starting to do something to finishing it.
- For user interaction, the latency from when a user provides input to the time they see the response is called lag.
- Controlling lag is extremely important for playability.
  - Too much lag distorts causality
  - When controlling motion, too much lag can make users motion sick.
  - Too much lag makes it hard to track or target objects.
  - High variance in the amount of lag makes interaction difficult
  - Users can more readily adapt to relatively high constant lag than to highly variable lag.

Computing lag



Reducing lag

- Pick a frame rate = 1/frame time
- Brute force: do as much as you can in a frame time
  - Faster algorithms and hardware means more gets done in that time
  - Budget your resources, everything – graphics, AI, sound, physics, networking, etc. has to be done in the frame time.
- Most important to reduce lag between user input and its direct consequences
  - Lag between input and other consequences may matter less.
- Update different parts of the game at different rates
- This means decoupling them
- What are the effective bounds on the frame rate?