Our Graphics Environment

Landscape Rendering
Hardware

- **CPU**
  - Modern CPUs are multicore processors
    - User programs can run at the same time as other programs
  - The level of parallelism (as enabled by the number of CPU cores) is relatively low
- **GPU**—massively parallel, set up for high-speed graphics
  - All modern high-performance graphics systems use massively parallel GPUs
- How do the CPU and GPU communicate? Slow data bus
Standard CPU process execution

- A 4-core CPU could run up to 4 processes simultaneously
- Each of the apps in this example may have many execution threads (processes), but at least 4 could run at the same time
- Each process reads and writes information to system memory using the system bus
GPU hardware environment

- The GPU is massively parallel (possibly thousands of cores)
- It has its own memory store and data bus
Programming on the CPU

- We write a program in some language (like Java)
- An interpreter or compiler translates our code to machine language
- The CPU reads our machine code and performs actions
  - Read and write memory, control other hardware, etc.
- The CPU talks to system memory along the system bus
Programming on the GPU

- Each GPU core is a computing unit, just like each CPU core.
- In our world of GPU computing, each GPU core will get the same set of instructions as all other GPU cores.
- But not necessarily at the same instant!
- This is called SIMD computing (single instruction stream, multiple data stream).
- As you will discover, parallel programming is interesting!
How do the CPU and GPU interact?

- All of our programs begin execution on the CPU
  - We will call that part of our program ‘the client’
- The CPU ‘launches’ the parts of our program that do rendering on the GPU
  - We will refer to the GPU as ‘the server’
- In almost all of our programs, we will occasionally need to move data from the CPU to the GPU
  - This is very slow! Do this as few times as possible!!
What does a GPU program look like?

- Each part of a GPU program is called a ‘shader’
- Different types of shaders are used for different parts of the ‘programmable pipeline’ on the GPU
- Our first programs will only have vertex and fragment shaders
- Others can have geometry, tessellation, and compute shaders
- In the end, we want to store numeric color data in the on-screen framebuffer
In practice...

- A vertex shader usually sets the rendered position of an object
- A fragment shader usually sets the color values for the pixels overlapped by the rendered object
The on-screen framebuffer

The on-screen framebuffer holds a numeric representation of the image.

The on-screen framebuffer holds a numeric representation of the image.

The display controller reads the framebuffer, pixel by pixel, and tells the monitor how to set the color of each display element.

Each cell is one of $W \times H$ pixels, laid out spatially. Each pixel holds a number (an RGB value).

The display controller reads each pixel’s numeric RGB value...

...and directs the monitor to color the corresponding location using the RGB value.
Thinking about GPU programming: vertices

- Assume you want to draw a rectangle like this:
- You can safely assume that each vertex will be assigned its own, independent ‘vertex’ processor
- Each vertex’s processor can be anywhere on the GPU
- Each vertex is independent
- There is no ‘edge processing’ here
- Edges are a result of rendering
Thinking about GPU programming: fragments

- The on-screen framebuffer is a set of pixels that stores the color of each portion of a rendered image.
- We will refer to these portions as pixels.
- A pixel holds a number representing a color.
- Each pixel’s color number is determined by its own independent ‘fragment’ processor.
- Essentially, each fragment’s RGB value ends up in the corresponding pixel of the on-screen framebuffer.
The display controller reads the framebuffer and renders the color corresponding to the RGB pixel value (using 0 and 1 in this example for simplicity) at the appropriate display element on the monitor.
Advantages of GPU programming

- Any shader program (vertex, fragment, etc.) runs simultaneously and independently on each processor.
- Each processor runs the same program but does so on different data.
- Processors generally don’t communicate with or wait for any other processors.
  - They can, if you really want them to, but you will pay a speed penalty...