Scribe Notes
CSCI 599: Physically Based Modeling for Interactive Simulation and Games (Spring 2011)
Topic: Simulation on programmable graphics hardware (GPUs)
Notes taken by: Samuel Woo
Date:03/21/2011

History of GPUs:

GPUs where originally powerful graphics processors used for rendering. Overtime they have become really fast and good at the task of rendering 3D graphics. In around 2001 the programmable pipeline was introduced in products lines such as nVidia's GF3 series.

Motivation:

After the introduction of the programmable pipeline people realized that they could use the power of a GPU for purposes other than rendering. One such purpose is to solve the motion of fluids using the Navi-Stokes equations. Some used GPUs because they could run certain workload faster than a CPU while others worked on the GPUs for the novelty of doing so.

Differences between CPUs and GPUs:

CPUs are designed to minimize the latency of sequential operations. This allows them to compute branching code very well. Branching software is has non-deterministic and non-predictable execution paths. It may not even be possible to determine when or if branching code will terminate. In order to handle these types of workloads, CPUs contain very complicated logic for branching while having only few computational/arithmetic components.

GPUs where designed primarily for rendering which has deterministic behavior. This type of work does not have branching, has a streamlined process, and have a predictable run time. Unlike CPUs, GPUs perform many of the same computation. They are SIMD processors that provide many computational units that do the same, deterministic calculation. While they do not handle branching very well, they can perform a single calculation on multiple data sets simultaneously. Therefore GPUs are fast for massively parallel programs and algorithms.

History of the (OpenGL) graphics pipeline:

Initially the graphics pipeline only provided for simple rendering. This included having transformations to a camera's perspective, rasterizing triangles, and everything necessary for simple 3D rendering. One other feature added later was the ability to multiply textures together in the rendering pipeline. This ability, called multitexturing, allows the combining of textures without pre-multiplying them and consuming extra memory for each combination. These additions to the pipeline are traditionally implemented in a process that starts with a hardware vendor designing an extension. Then having the extension adopted by multiply venders, and finally having the extension approved by the Architecture Review Board and having it implemented as a standard feature into the library.

OpenGL 2.0 and Shaders:

Prior to shaders only the features adopted as standard where available in the graphics pipeline. Shaders added the ability to modify the pipeline by implementing a custom computation. This allowed for many new effects to simulation complex materials, improve lighting and shadowing, and doing more advanced texture mapping. These shaders also allowed people to implement non-graphics related computation by implementing a custom computation and writing the results to a buffer unrelated to graphics.

The Graphics Rendering Pipeline:

1. CPU – Generates vertices.
2. Vertex processor – Determines vertex positions. Is customizable by vertex shaders.
3. Rasterizer – Determines pixel coverage of triangles.
4. Fragment Processor – Calculates final color (using texture mapping etc.). Is customizable through pixel shaders.
5. Frame Buffer – Computed values placed here. Is not the same as a monitor since a monitor is not storage than can be read.

Evolution of GPGPU:

Initially the fixed function graphics pipeline could not be used for general purposes. The introduction of programmable shaders allowed for general purpose computation; however anyone doing GPGPU work had to understand the graphics pipeline so that they could get their data to the shaders that they had customized. Essentially one had to pretend to do graphics in order to use the GPU for general computation. To address these issues nVidia introduced the CUDA language that hid the graphics pipeline and replaced the shader language with the standard C language. OpenCL is later developed which provides similar functionality as CUDA, but is supported by multiple vendors.

Data Bandwidth Issues For GPUs:

In order to do GPGPU computation data must first be pushed across the bus connecting the CPU to the GPU. Data will also need to cross this bus to be returned and used by the CPU. The bus is very slow (devices are at least few centimeters apart) and is often a limiting factor for rendering. However rendering only requires data to be sent to the GPU. Reading back data from the graphics card is even slower.

One recently added shader to the graphics pipeline is the geometry shader. This sits between the vertex shader and the rasterizer and allows the creation of new geometry in the middle of the graphics pipeline.

Questions asked from class:
Q: Due to the closeness of an Integrated GPU to the processor can integrated GPUs be faster by eliminating the bus?
A: Integrated graphics have always been weak processors that lack many features. They have been the "poor man's" graphics.
Q: When one places a custom program/shader into the graphics pipeline, does this affect other programs that may be using the pipeline?
A: Modifications done to the GPU are specific to each application. Different modifications can run at the same time and the graphics pipeline can be changed while it is in use. There are commands that programs can use to change shaders while running and different applications will always use their own shaders and modifications to the pipeline.

Summary of Shaders:

Pre-2004 shaders where programmed in assembly. Afterwards OpenGL implemented the GLSL and Microsoft's created the HLSL languages that allowed programming shaders in a language similar to C.

The Vertex Program:
Input: vertex properties including position, color, normal, texture coordinates, etc.

Output: position in clip coordinates vertex color, normal, etc.
Example Program: Displacement mapping – add the displacement texture to the vertex position

The Fragment Program:
Input: pixel attributes including color, normal, texture coordinates, and many more possible attributes
These values are computed by the rasterizer and computed from the interpolated vertecies
Output: color, depth
Example Program: multitexturing

Features of the GLSL:

- Includes many properties that can be retrieved by the shader
- Floats: 32bit, but they are not fully IEEE-754 compliant (they introduce some error but ok and not visible in rendering)
- Integers: at least 16 bit integers
- Booleans
- Vectors of 2, 3, 4 elements
    - Standard arithmetic operators used on vectors are done component wise
    - Supports swizzling, or taking subsets of the vector as a new, smaller vector
- Matrices of 2x2, 3x3, 4x4 elements
- Sampler type used to access textures
- Global Qualifiers: attributes that do not change between different vertices and attributes that remain the same across all executions of the shader
- Loops: C++ style if, else, for, while, do loops
    - Using loops may negatively impact shader performance
- Built in functions
    - Trigonometric
    - Exponential
    - Common math functions
    - Geometric
    - Relational

Questions asked from class:
Q: Is it better to do multiple small shaders or use a longer branching shader?
A: If we know that all executions of the shader will follow the same branch then one should use multiple shaders. Otherwise, if the branching can't be pre-determined and will be different for each execution you have to create a larger shader with branching. However, shaders are supposed to be very short programs and may not even compile if they are too long.
Q: Is there an efficient way to debug shaders?
A: No. The only way to debut shaders are to give it input and inspect the output. Then do further experiments comparing the inputs to the outputs. This is painful and slow.

Doing A Mass Spring Simulation in GLSL:

- Store the physical state into textures
    - First pixel stores position of first vertex. Each pixel stores one float.
    - Fill the texture in linearly until you have entered all information. Any extra pixels in the last row can be ignored
    - Store position, velocity, and acceleration into the textures using this method
    - Create a separate texture to store vertex to spring mappings

- In the fragment program
    - Render full screen quad so that the rasterizer splats the quad across the screen
    - Each pixel will receive a texture coordinate from the rasterizer and this will map to each spring force in the texture storing them
    - Lookup the vertices from mapping and compute the spring force from the position and velocity textures
    - Render these results to a texture that contains all the spring forces
- Run a second pass through the fragment shader that takes the spring force texture and combines it into the acceleration texture