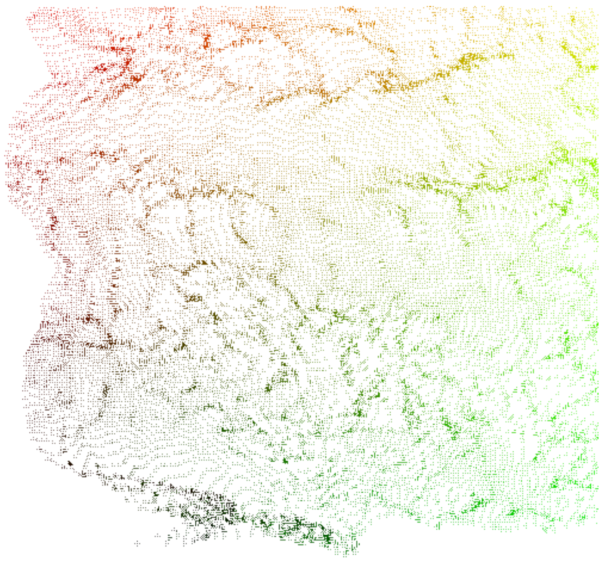
GPU-Based Objects with Collision and Physics

Mentor
Dr. Kalpathi R. Subramanian

Research and Developed
Jason S. Hardman



… science and art upon the aether – flatline0

Table of Contents

Abstract

Summary
The current generation of gaming platforms includes phenomenally fast GPUs (Graphics Processing Units) that allow the developer to program directly on the GPU. This technology is called Shader Programming. Far from being a simple "airbrushing" technique, shaders have all the necessary components for doing General Purpose computations on the GPU (GP-GPU). With the speed of the GPU and its reputation for beating Moore's law, this technology opens a doorway into a new paradigm of processing

We have developed a reusable language framework to encourage growth in the area of general GPU programming. Shader computation can be difficult to get a design perspective on. We hope to facilitate the community with some ropes and pulleys in an effort to make a "c++ to c"-like extension to tame the quirky style of gpu computation.

Background
We propose a GPU-based Object Oriented Framework and provide examples of usage through examples of collision and physics. Our research is from two independent test beds. We decided to footprint the GPU with a low-overhead harness to find out what organization system best resonated with the hardware. After some redesign, we built the system out as an attachment to a pre-existing game engine to demonstrate its usage.

glCompShader
Our statistics and bottom-up research comes from a harness program written in C++ with OpenGL and GLSL (OpenGL Shading Language) for the purposes of accurately measuring the capabilities of the hardware. We want to determine the most optimized frequencies for accessing memory and running batches of shader instructions.

Ele*Mental* Game Engine
Our theory and top-down research comes from an engine written in C# and DirectX using XNA ("XNA's Not Acronymic") with HLSL (High Level Shading Language). We use it to show the real world applications of our design as integrated with standard industry technologies.

Conclusions
By the conclusion of this paper, we have began to see a several design patterns emerging. A basis for for sub-object encapsulation on the GPU exists by moving parts of CPU-objects across the bridge entirely. List based processing, using Spatial algorithms for interaction, handles a variety of visualization problems well and has design theories in common with listener event handling.

We've had several problem reduction strategies emerged by treating the GPU as a multi-dimensional signal processor. Splitting data by variable helped us organize data in a List approach. Point Rendering provides a two-dimensional algorithmic solutions basis for organization. Facilitated by our two-way pointer system we encountered reduced problems with scatter/gather.

We propose a variant our framework as a worthy next step beyond the geometry shaders of Shader Model 4. Providing per-class shaders in hardware would allow for a natural interface to stream programming by establishing a one-for-one equivalence between the language semantics of CPU and GPU code. Then, CPU languages can be programmatically converted to GPU languages using a natural extension to the current industry language trends. An extension, that is also intuitive to an Object Oriented Industry.

Acknowledgments

Our research is ultimately an extension to the wealth of information found at **www.GPGPU.org**. Their forward research, projects, and tutorials delving into the subject has been indispensable to the theoretical bases of the implementation of this system. Anyone serious about 3D-Engine design and implementation would do well to read or review.

Thank you to my **Mentor, Dr. K.R. Subramanian** whose continued issuance of rope over the last two-years of this one-year project has given me the freedom hang myself several different ways and readjust. .Also, I must commend him for his tolerance of my habit of designing every detail in my head before writing it down.

I can't explain how helpful my **Assistant Coder and Review, Subhir Rao**, has been for pulling together some of the integration of this algorithm. He has been my sounding board for “thinking in hardware” more than he realizes and I’d have had a helluva time of it without his assistance.

A big thanks to my **Technical Documentation Assistant, Amanda Chafin.** This document is more like English now and less like pseudo-code thanks to her efforts.

We would like to thank the **EleMental Engine Project at** www.darkwynter.com for providing a C# testing environment that is flexible and robust enough to handle the requirements of our framework. Without it, we still probably wouldn't know if any of this actually works. We look forward to their upcoming IGF competition bid in 2008-9.

We want to thank **The Charlotte Regional Visualization Center** and **Games and Learning Lab,** both at **UNC-Charlotte**. They have been excellent academic resources.

We want to thank **Neon Helium** and **Riemers.net** for their services to the community. Anyone interested in learning more about graphics programming should search the web for these two excellent sites.

We would like to thank **XNA** for providing a light-wait, user-friendly platform that handles asset loading like a dream and provides very non-intrusive access to the GPU. Great work to the team and best wishes to the home-brew nation.

Additionally, I must extend a thank you to the general **Graphics Community**. I have six gigabytes of text data from two years worth of research that has “probably” not been used directly in this paper. Many ideas have influenced its design and I apologize if I have left you out. If I’ve missed your inclusion or if you have comments, suggestions, reviews, or projects please contact me.

After our beta development effort, we intend to release a dll for academic. At this point, we'll fill in the necessary citations and documentation so that the community can validate our theory for themselves. http://darkwynter.com/flatline0.

Thank You All
:-j

§1 - Proof of Concept Experiment

1.1 - Summary

The focus of the experiment was to find the optimum frequencies for running GPU calculations at practical frame-rates. This involves balancing the texture size, the number of passes, and the frame rate in a way that most optimally uses the hardware to produce the greatest number of Instructions per Second.

1.2 - Hypothesis

Current industry usage would suggest that the GPU is most efficiently organized to handle 128x128 and 256x256 textures.

If so, then we may be able to organize a more intelligent data structure to eliminate scatter from many of our calculation sequences by organizing out data in a way that maximizes computational organization and is less tied to spatial organization.

We would like to find out if it is also optimized to handle many short programs as or more efficiently than running long elaborate programs.

If so, we can construct atomic operations that run in sequence, while outputting the intermediate results of their calculations into textures. In effect, we want to create re-programmable assembly instructions.

1.3 - Procedure

Our test environment is glCompShader, a bare-bones program that was written in C++ with OpenGL and GLSL (OpenGL Shading Language) for the purposes of accurately measuring the timing of the hardware. We use the NVIDIA 6800 card with IDE-Express bus architecture as our testing benchmark.

To test the efficiency of both quad and point based rendering, we wrote a simple shader algorithm to make a single operation, per color channel, per pixel, per pass. The visual effect is a repeating pattern of colors if visualized on screen.

Experiment Inputs Variables:

We changed the size of the Render Target texture in 2^n multiples.
We changed the number of shader passes executed on each CPU pass.
We executed four operations per pixel, or one per Color channel.

Experiment Output Variables:

Instructions per second	= Width x Height x Passes x Colors x Frame-rate
Instructions per frame	= Instructions-per-second / Frame-rate

Environment - glCompShader

A minimalist GPU harness written in C++ with OpenGL and GLSL.
Tested on NVIDIA 6800 IDE-Express
CPU requirements negligible

1.4 - Data

Using Quad Based Rendering

Width	Height	Passes	Colors	Frame Rate	Instructions Per Second	Instructions/Frame
64	64	256	4	28	117,440,512	4,194,304
64	64	512	4	25	209,715,200	8,388,608
64	64	1024	4	18	301,989,888	16,777,216
64	64	2048	4	12	402,653,184	33,554,432
64	64	4096	4	8	536,870,912	67,108,864
64	64	8192	4	4	536,870,912	134,217,728
128	128	128	4	25	209,715,200	8,388,608
128	**128**	**256**	**4**	**25**	**419,430,400**	**16,777,216**
128	**128**	**512**	**4**	**18**	**603,979,776**	**33,554,432**
128	**128**	**1024**	**4**	**12**	**805,306,368**	**67,108,864**
128	**128**	**2048**	**4**	**7**	**939,524,096**	**134,217,728**
128	**128**	**4096**	**4**	**4**	**1,073,741,824**	**268,435,456**
128	128	8192	4	2	1,073,741,824	536,870,912
256	256	32	4	25	209,715,200	8,388,608
256	**256**	**64**	**4**	**25**	**419,430,400**	**16,777,216**
256	**256**	**128**	**4**	**18**	**603,979,776**	**33,554,432**
256	**256**	**256**	**4**	**12**	**805,306,368**	**67,108,864**
256	**256**	**512**	**4**	**7**	**939,524,096**	**134,217,728**
256	**256**	**1024**	**4**	**4**	**1,073,741,824**	**268,435,456**
256	256	2048	4	2	1,073,741,824	536,870,912
512	**512**	**16**	**4**	**25**	**419,430,400**	**16,777,216**
512	**512**	**32**	**4**	**18**	**603,979,776**	**33,554,432**
512	**512**	**64**	**4**	**12**	**805,306,368**	**67,108,864**
512	**512**	**128**	**4**	**7**	**939,524,096**	**134,217,728**
512	**512**	**256**	**4**	**4**	**1,073,741,824**	**268,435,456**
512	512	512	4	2	1,073,741,824	536,870,912

Figure 1.1: Instructions per Second for Quad Based Rendering

Quad Based Rendering - Mapping a texture to a 4-vertex square, then Rendering To Texture using an orthographic camera. This is the standard, and fastest way to compute using the GPU.

Key:

Yellow	**= Optimum range for gaming applications**
Bold	**= Optimum range for scientific applications**

Using Point Based Rendering

Width	Height	Passes	Colors	Frame Rate	Instructions Per Sec	Instructions/Frame
32	32	1024	4	15	62,914,560	4,194,304
32	**32**	**2048**	**4**	**10**	**83,886,080**	**8,388,608**
32	**32**	**4096**	**4**	**6**	**100,663,296**	**16,777,216**
64	64	64	4	25	26,214,400	1,048,576
64	**64**	**128**	**4**	**25**	**52,428,800**	**2,097,152**
64	**64**	**256**	**4**	**18**	**75,497,472**	**4,194,304**
64	**64**	**512**	**4**	**12**	**100,663,296**	**8,388,608**
64	***64***	***1024***	***4***	***8***	***134,217,728***	***16,777,216***
64	64	2048	4	4	134,217,728	33,554,432
128	***128***	***32***	***4***	***25***	***52,428,800***	***2,097,152***
128	***128***	***64***	***4***	***18***	***75,497,472***	***4,194,304***
128	***128***	***128***	***4***	***12***	***100,663,296***	***8,388,608***
128	***128***	***256***	***4***	***8***	***134,217,728***	***16,777,216***
128	128	512	4	4	134,217,728	33,554,432
128	128	1024	4	2	134,217,728	67,108,864
128	128	2048	4	1	134,217,728	134,217,728
256	256	4	4	23	24,117,248	1,048,576
256	256	8	4	18	37,748,736	2,097,152
256	256	16	4	12	50,331,648	4,194,304
256	256	32	4	7	58,720,256	8,388,608
256	256	64	4	4	67,108,864	16,777,216
256	256	128	4	2	67,108,864	33,554,432
256	256	256	4	1	67,108,864	67,108,864
512	512	2	4	15	31,457,280	2,097,152
512	512	4	4	10	41,943,040	4,194,304
512	512	8	4	7	58,720,256	8,388,608
512	512	16	4	4	67,108,864	16,777,216
512	512	32	4	2	67,108,864	33,554,432
512	512	64	4	1	67,108,864	67,108,864

Figure 1.2: Instructions per Second for Point Based Rendering

Point Based Rendering - Mapping a texture to a set of points (one point per pixel), then Rendering To Texture using an orthographic camera. Each point can be moved in the vertex shader, making this slower rendering method more flexible for spatial algorithms.

Key:

Yellow	**= Optimum range for gaming applications**
Bold	**= Optimum range for scientific applications**

1.5 - Analysis
The memory architecture for our NVIDIA 6800 GPU seems to handle Quad based rendering at approximately 8x the speed of our Point based rendering system.

Figure 1.1 shows that the Quad based rendering system works best with 128 and 256 square textures. We find our best results using 16 to 1024 passes, with more passes providing better efficiency.

Figure 1.2 shows that the Point based rendering system works best with 64 and 128 square textures. We find our best results using 32 to 512 passes, with more passes providing better efficiency.

These results are consistent with the idea that keeping the graphics pipeline as full as possible, verses thrashing it with data, yields the best results.

1.6 - Conclusions
We conclude that a philosophy of using many short shader instructions in sequence is a viable method of code organization. Our results show that by using a larger number of passes in our computation, we are taking the most optimized approach to data processing supported by current industry hardware.

Our development process should then be to dissect complex procedural equations into atomic shader units, which saves their values into texture memory. These atomic units are analogous to an “equal” sign of an equation. The variable that is being assigned to is stored in a texture while the intermediate results of the equation are inaccessible outside of that equation.

By saving intermediate results to atomic equations, we significantly reduce the impact of GPU-programming scatter/gather problems and make the construction of our application simpler to manage and modify. In theory then, this gives us a concept of global memory within the shader-programming paradigm.

The net result would allow us output resultants intermittently while executing a longer instruction sequence. This means we can define an abstract methodology for dissecting large or interwoven algorithms. This gives us a natural way to translate programmatic expressions into shader components, a system which can potentially be automated.

§2 – An Object Oriented GPU Framework

2.1 - Summary

We provide a GPU data framework written in C# and HLSL using XNA.

We provide a third-level language framework for doing general computations on the GPU by creating an Object Oriented layer on top of the procedural style code of GLSL and HLSL. Also, we want to expand on current code porting techniques to facilitate movement of complex algorithms from the GPU so that CPU requirements, like AI, have room to expand.

Our example demonstrates a collision and physics build-out procedure that is configurable to share data with standard draw cycle shaders. This provides a closed-loop system for GPU-Based objects, with both draw, update, and the notoriously complicated collision mechanisms no CPU oversight.

A subset of our world-objects' properties reside on the GPU and require no transference to the CPU for reorganization. Thus we maximized computation while avoiding CPU-GPU bridge costs.

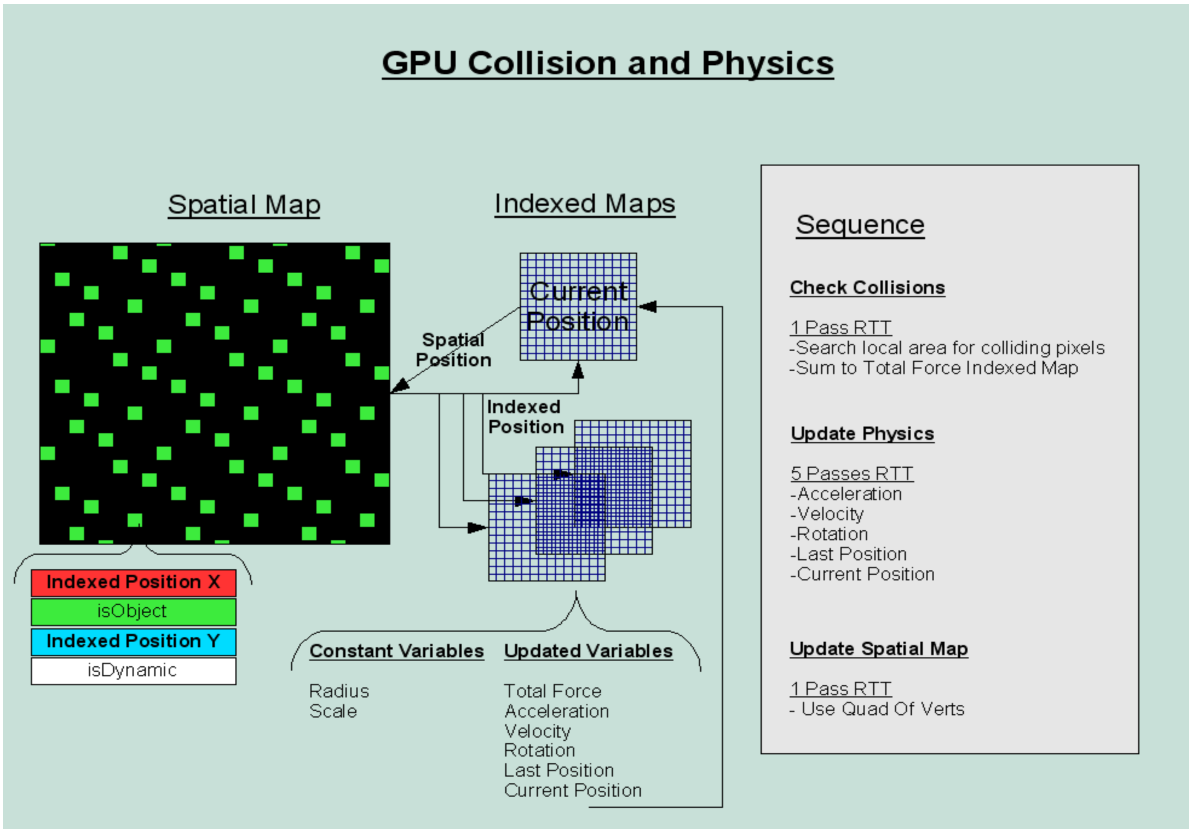


Figure 2.0: Our algorithm requires two methods of data storage that we refer to as Spatial Maps and Indexed Maps. A seven-pass sequence updates each of the variables in our physical motion formula, then the spatial data structure.

Our framework is composed of two data structures and a system of pointers:

- The Spatial Maps are used by the Collision algorithm to access surrounding objects.
- The Indexed Maps are used by the Physics algorithm to update object properties.
- A two-way system of "Pointers" is used to link between the two structures.

2.2 - GPU Object Encapsulation

A consequence of moving collision and physics to the GPU is that the positional variables required by the draw cycle remain in GPU memory. Using our Framework in combination with Vertex Buffer Objects puts virtually the entire object on the GPU.

This reduces the amount of data passing through the GPU-CPU interface, which is a significant bottleneck. A PCI-Express Bus (x16), for example, runs at 8 GB/sec. In contrast, the GPU memory interface for a GeForce 6800 GPU is 35 GB/sec. Also, it is interesting to note that the CPU- Memory Interface (800 Mhz Front-Side Bus) is slower than both, at 6.4 GB/sec. (GPU Gems 2 – Table 30.1)

By moving the majority of our object properties to the GPU, we can compute realistic physical interaction for tens-of-thousands of objects while freeing up CPU computational cycles.

2.3 - RISC Approach to GPU Computation

Taking the results of our previous experiment, we divide our code into atomic operations to maximize code reuse and circumvent parallelization issues. This allows us to process a large numbers of objects in parallel while maximizing our ability to intercommunicate among streams of data. Self-referencing equations are handled through the use of duplicate textures. Each texture holds a single property for a collection of objects. Our approach is highly extensible, since writing atomic code units is trivial. Expansion of the architecture to include libraries of shaders is made easy.

2.4 - Spatial Maps

This algorithm is visualized as “shot-gunning objects”, where each shot-pellet re-directs to a particular object's location and stores its corresponding Indexed texture coordinate there.

The Spatial Map is a two dimensional array, stored in a texture, which represent an object's position in 2D space. Each position, or pixel, stores a texture coordinate in the R and G channels. These positions point to the object's properties, which are stored in the Indexed Maps. All empty positions are assigned a 0.0 value.

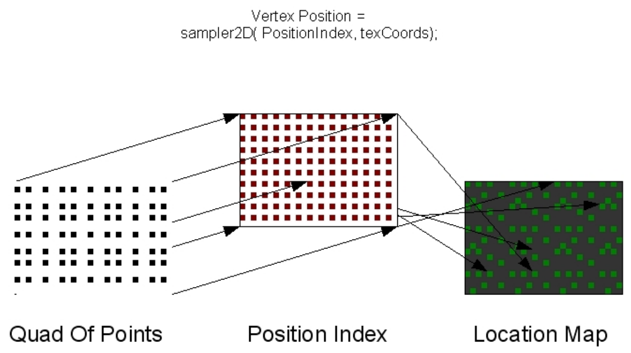


Figure 2.1: Each vertex gets its position from the Indexed Position space it matches. This spatial coordinate is its position in space, but stored in a set of 2D texture

The Spatial algorithm is the trickiest part of the design. It cannot be accomplished in the Fragment Shader because it would require pixels to access other pixel values in order to move an object from one location to another. To solve this problem, we use the Vertex Shader, RTT (Render To Texture), and a quad of points to move objects from one location to another.

The quad of points must contain one vertex for each texture coordinate in the Indexed Maps. For each vertex, we get the object's position from the Indexed Position Map and assign it to the vertex's position. Then we set the R and G channels to the vertex's Indexed (or original) texture coordinates. This provides a pointer system, or a way by which we can trace from the Spatial Map back to the Indexed for any object. The result is a texture that contains a point for each object on a blank background.

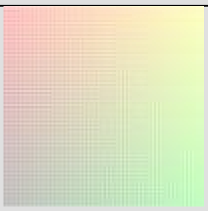
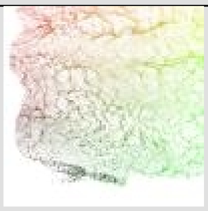
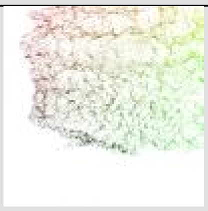
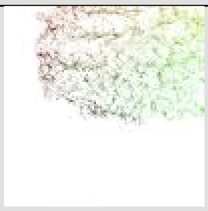
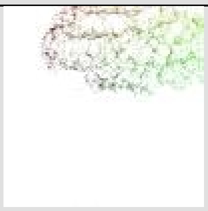
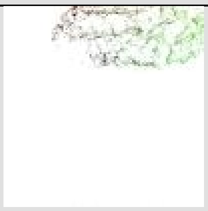
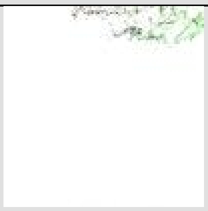
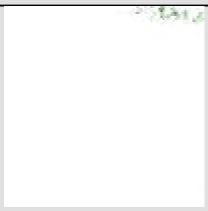
Spatial Map in Motion			
 dt = 0	 dt = 1	 dt = 2	 dt = 3
 dt = 4	 dt = 5	 dt = 6	 dt = 7

Figure 2.2 : First 7 frames of a Spatial Map as pixels representing objects move in 2 dimensions.

We use the A channel of the Spatial Map as a binary flag to indicate that a position is occupied by an object. This is done to avoid confusing a 0.0 texture coordinates with an empty location. This usage is non-optimal, however, it is useful for debugging and simplifies the algorithm.

2.5 - Indexed Maps

Overview

A collection of Indexed Maps is best through of as a GPU-Object. The GPU-Object acts as a wrapper for several GPU variables and contains methods to facilitate the creation of GPU variables. In other words, a single "GPU Object" represents a list of objects belonging to the same class definition on the CPU. For our physics and collision algorithm, we defined the following GPU-Variables:

```
public Vec3 currentPosition
public Quat rotation
public Vec1 scale
public Vec1 mass
public Vec3 velocity
public Vec3 acceleration
public Vec3 lastPosition
public Vec3 totalForce
public Vec1 radius
* public Vec3 dummy
```

**Note: The dummy variable contains trivial data in its textures and is required because of an unknown bug in XNA GSE 1.0. In the final pass of the algorithm, we would sporadically get a completely black texture. Appending the dummy texture to the end of the sequence catches the black texture so it doesn't affect calculations.*

Initialization
The GPU object constructor initializes each GPU data type.

Adding Objects
Since a single "GPU Object" represents a large list of CPU objects, a list of CPU-objects must be passed in from the application. To retrofit to our existing engine, we loop through each CPU-object calling this method and passing in it's properties to the corresponding GPU variable.

Finalizing Objects
Once all properties of all CPU-objects are added, we call finalize on each variable, passing in the minimum and maximum values that the data spans.

2.6 - Pointers

Two data formats are linked to each other using a system that imitates pointers. We use this system to easily convert between our Spatial and Indexed data organizational system.

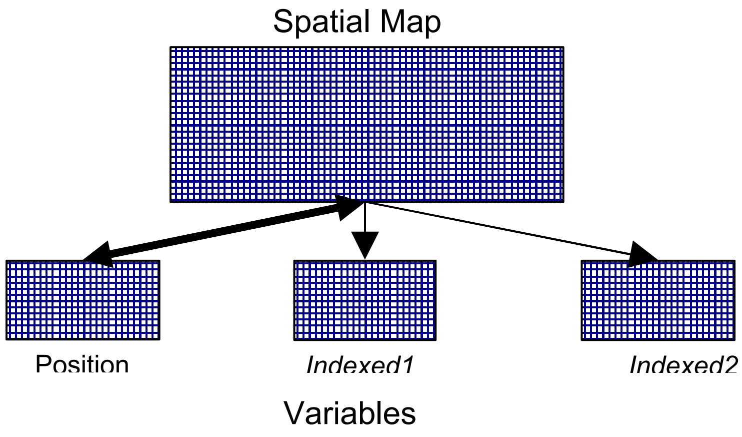


Figure 2.4: A circular system of pointers connecting Spatially organized data and Indexed Variables. An object's location is marked in the Spatial Map using texture coordinates. These texture coordinates act as pointers back to the object's properties, which are stored in the Indexed Maps. The Position Indexed Map points back to the object in the Spatial Map.

The Spatial Map pixels contain Position texture-coordinates, while Position pixels store texture-coordinates for location in Spatial Map. This circular pointer system allows an Indexed shader program to access a neighboring object. The pointer works as a key to access the object's Indexed properties.

An Indexed Map object can also access the Spatial Map by using the data contained in its Indexed Position variable. The Indexed Position is also the object's spatial coordinates. From there it can search neighboring pixels of the Spatial Map for nearby objects to act on.

2.7.1 - Variables – Design Theory

Overview
A GPU variable class takes a list of like-properties and converts them into a format the GPU can use.

Indexed Maps store object properties. Each object is assigned a unique texture coordinate which identifies an object across multiple textures. This texture coordinate is stored at the object's position in the Spatial Map.

Each texture holds a single like-property from multiple objects. The underlying texture size is set to the expected number of objects. For applications needing to delete and recreate large numbers of objects, we suggest indexing the objects and tracking deletions. This combined with an intelligent shader would allow for de-fragmentation checks per pixel for neighbors. For example, a Point-based shader can be configured to shift a vertex carrying with it all the information for the underlying CPU variable. By de-fragmenting, areas of memory can be filled with new objects.

Figure 2.3 Position Indexed Map on uneven terrain

<u>"Ping-Ponging" Data (1)</u>
A texture can not access itself and render to itself on the same pass. Or rather, doing so can result in undefined behavior by the hardware. Therefore we must keep two copies of each texture, a read copy and a write copy. This provides us with a working copy and a clean copy of the data to use during execution phases. "Ping-Ponging" is analogous to the way the frame buffer operates.

This system also helps us for any equation that must read its own value (e.g. +=, -=, /= operations). By using the write copy as our Render Target, and using the read copy for any self referencing equations, we can work around this complication. At each pass, the working set of read and write textures must be swapped to provide the last passes values to the current pass.

<u>Accessing Data from CPU</u>
Data can be read/written from the CPU using sub-texture operations. Intelligent sampling should be used to tunnel for values using the texture mip-maps.

<u>Alternative pixel storage method for 64-bit Data</u>
A limitation of data storage in textures is that, for the most part, they only hold data in a range from 0-1 and precision is limited. All formats must support 8-bits per channel to contain color data. For high precision applications, we can subvert this limitation by using the R, B, G, and A channels as individual bytes of a four-byte word, or 64 bits. This gives us a natural mechanism for guaranteed 64 bit processing on the GPU.

<u>Memory De-Fragmentation</u>
Memory de-fragmentation can be handled by marking dead objects with a null identifier (e.g. – 0.0) in one of the texture channels. Garbage collection is handled in the vertex shader with a simple recollection algorithm.

For example, a quad of points algorithm (see Spatial Maps) can be used to reorganize Indexed Maps. Each vertex checks its neighbors and shifts it's location if empty. Over time, the data steadily migrates towards one side of the texture, leaving empty space for object creation on the opposing side.

2.7.2 - Variables – Interface Methods

Defined Data Types:

Vec1 = 1D Vector	Stores data in R channel
Vec2 = 2D Vector	Stores data in R, G, channels
Vec3 = 3D Vector	Stores data in R, G, B channels
Vec4 = 4D Vector	Stores data in R, G, B, A channels
Quat = Quaternion	Stores data in R, G, B, A channels

All Data Types Include:
2 Render Targets and 2 Textures
A reference to the GPU shader (Effect or Program)
A reference to the GPU texture our variable maps to.

Initialization
We create both Render Targets and both Textures. OpenGL applications should only need a single render target for each sized texture being used. For our XNA implementation, we had to attach a render target to each texture due to some undefined behavior with texture pointers coming back from the GPU. We need two textures per variable because the shader requires both a read and a write texture for operations that contain self-referencing equations. In addition, we pass in a reference to the shader program, map the CPU texture to the appropriate GPU texture sampler, and store a user defined name for the variable.

Adding Properties
A group of objects contains the same properties across them. A GPU variable represents the same property across groups of objects. To convert between the two organizational strategies, we first add all the objects to a private list owned by that data type. Once all objects have been added, we still must finalize the operation.

Finalizing a Data Type
Once all values have been added to the private list, we convert the list to a texture format. Based on user input, we calculate the range of data and scale the values between 0 and 1. We then add the data to both the read and write textures to avoid confusion.

Swapping Buffers
This method must be called once per update cycle. It takes the read and write textures and texture buffers and swaps them. This allows us to keep the most recently calculated data in the Read texture, while always overwriting the Write texture. This is also known as The Ping-Pong Technique(1).

Getting Data Back
We will, most likely, want to sample the contents of the GPU variable from the CPU. This is done by a method that un-scales the data and returns the value requested. Ranges of data can also be sampled by using sub-texture access methods.

Writing To File
This is included as a useful debug method. This method is VERY expensive, so be sure all calls to it are removed after debugging. It gets both textures from the GPU and writes them to file in the running directory, appending Read and Write to the variable name set by the Initialization method.

§3 - Technology Application and Results

3.1 - Summary

The following is a description of how to create a physics and collision architecture on the GPU. We begin with textbook Linear Motion Physics. Next we add a "Wind Map," which will add location based wind forces to the objects. Lastly, we discuss methods of implementing GPU-based collision detection for 2D engines, 3D engines, and large environment engines.

3.2 - Linear Motion Physics

For the physics update, we use the traditional method of rendering data to a quad using the fragment shader. The difference is in our method of breaking down sequence of operations atomically.

The method's operations are partitioned wherever a value must be saved for use in the further calculation (e.g. velocity, force, location, and rotation). Then, for each atomic operation, we use a separate pass and RTT (Render To Texture) to store the values in the appropriate Indexed Map.

Figure 3.1: Initialized Position Variable for 256x256 CPU Objects

Our previous research shows that we can trade off between using a large number of passes to execute short operations and just a few passes to execute long shaders.

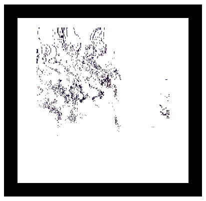


Figure 3.2: Initialized Acceleration Variable for 256x256 CPU Objects

This technique enables us to take an algorithm and provide intermediate values that are accessible by other shader objects. This provides solution to one of the limitations of GPU computing, being the inability to reference information from another pixel or vertex while in mid computation. By saving intermediate values to textures, the next atomic pass can read the values of any other object by reading the previously rendered texture, thereby circumventing this limitation.

3.3 - Fluid and Wind Simulations

The following Wind texture contains three color-channels, representing a force vector at each pixel.

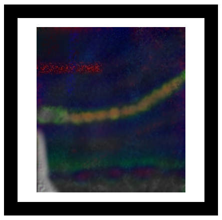


Figure 3.3: A 3-channel vector map overlaying terrain to create currents

When calculating the total force map, we reference the Current Position map to determine where the object is in world space. Using this position, we read the Wind Map at the object's location and apply the vector force found there. At the end, we add it to the Total Force Indexed map value.

Modulating between multiple Wind Maps at different elevations provides a light-weight solution to creating realistic wind and water current using simple texture blending techniques like height based averaging.

Using one Wind Map for each axis creates a create a lightweight voxel-based force map.

Include sinusoidal, fractal, or volumetric noise to animate the force maps and provide more realistic and dynamic environments.

After stress testing our implementation we noticed that our object have a tendency to group together after, say ten minutes of continuous processing. Reloading the objects is a simple fix for this. Implementations that cannot reload objects without causing visual incongruity need collision detection.

3.4 - Collision Detection

3.4.1 – In Two Dimensions

Collision detection is computed during the Total Force update pass. For every object in the Total Force map, we reference the Spatial map using the texture coordinates stored in the object's Position texture. This gives us the object's location in 2D space where we can search surrounding pixels to find potentially colliding objects.

When we find an object within the search region, we branch off to complete a mathematical detection of the collision using the two objects' radii. Colliding object properties can be found by accessing the texture coordinate values stored in that object's Spatial pixel.

After all the collisions have been detected, we sum the final result and render it into the Total Force Indexed Map.

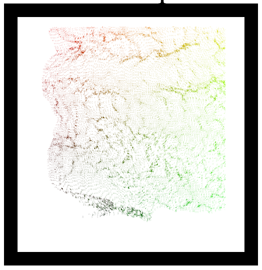


Figure 3.4: Scattering objects (e.g. - leaves) in the Spatial Map after applying the Wind Force Map

update.

3.4.2 – In Three Dimensions

Since the Spatial Map uses a 2D texture to store positional data, it is possible for multiple objects to overlap each other or occupy the same pixel without colliding. For many applications, simply using a fine enough pixel to world-space ratio would provide sufficient detection . Remember, that the collision search area must be adjusted accordingly to account for low frame rate collision misses.

Another solution to the problem comes from the fact that the limitations of our 2D algorithm come from having only two dimensions. The width and height of the texture leaves us without a third access to detect other objects. Adding another Spatial Map, from a different axis, solves this problem. In our collision detection algorithm, we then sample the surrounding pixels of two different maps to ensure that we catch any collisions that might be missed from another angle.

3.4.3 – Massive Environments

Our collision algorithm requires discrete pixels or voxels that are smaller than the size of the object they represent. If not, the objects will write to the same voxel on all axis and collision will be missed. This limits the area that any one Spatial Map can cover without loosing collision precision.

Spatial Maps are most efficiently calculated using 64x64 or 128x128 sized textures, and support a larger number of passes This means we can calculate a series of Spatial maps that sub-section an area, without impacting the frame rate to heavily. We also recommend using card-supported LOD (Level of Detail) levels provides a natural way to reduce calculation by searching higher LODs first.

3.4.4 – Comparison to 3D Textures

Its been noted that the same functionality can be achieved by using 3D Textures to hold object references. This is indeed possible and probably simpler to implement. The GPU memory requirements for storing data in a 3D array is astronomical however, and is not an option for many applications due to other demands for that memory.

§4 - Conclusions

This paper summarizes two years of research into the GPU by the author from the perspectives of both OpenGL's GLSL and DirectX's HLSL.

As far as the two shader languages are concerned, they are similar enough to require very little discussion. Both GL and DX provide a full interface to the GPU that allow unimpeded control over the hardware.

We like DirectX specifically for its interface organization and relatively easy asset handling. For industry applications, however, we still prefer OpenGL for its inclusion of VBOs (Vertex Buffer Objects), a set of commands that allow vertex caching in GPU memory. In an environments that can support hundreds of thousands of objects, the bottleneck has shifted to the raw number of vertices that can be passed across the bridge. OpenGL solves this with VBOs by only transferring the vertices across the bridge once. It is unclear how exactly DX is storing these buffers, as documentation is unclear.

In our opinion, GPU and Cell based processing is leading the industry into another paradigm shift in computation. We have seen one dimensional command input, two dimensional windowing systems, and now the capabilities exist for three dimensional systems to evolve. We have felt for some time now, that the CPU is best suited for management and intelligent systems and should remote administrate the functions of the GPU, which is better suited to handle what we code-hackers call “brute-force” work.

In the world of visualization and games, the “brute-force” work is traditionally found in graphics algorithms and physics calculations. The pursuit of better graphics has seen remarkable improvement over the last ten years, but is limited by the pure number of real calculations you can do per object to make it interact with its environment realistically.

By providing an extensible framework for handling large scale object management by offloading to the GPU, we provide a mechanism by which fully intractable environments can be created, while at the same time free up CPU clock cycles to often overlooked elements of game play like advanced Artificial Intelligence systems.

We hope to inspire a change in the way games are played as a whole. Player attack methods can include interactive high-density particle systems. Lakes and sand pits can be constructed from realistic particles and applied in ways that two-dimensional sprites cannot accomplish. Wind and water force maps provide realistic flight and naval challenges such as changing currents and wind gusts. We would like to see this system used to provide bend-forces to grass in fields, real motion to leaves in trees and the wind, and flocking patterns of birds, fish, and insects. We have seen evidence during the course of development that all of these are possible with this framework.

All concepts discussed in this paper have been proven, though there are certain scaling and normalization issues that we are working on to abstract the framework for general consumption. After this development effort, we intend to release an API so the community can validate our framework for themselves. We hope this document proves useful and invite you to contact us if questions arise at **http://darkwynter.com/flatline0/compSci/gpuPhysics** .

§4 – Citations

1) Dominik Göddeke -- GPGPU::Basic Math Tutorial