Interactive Simulation and Visualization using the GPU

Torre Zuk[ab*] and Sheelagh Carpendale[a]

[a] University of Calgary, Department of Computer Science, Calgary, Canada;
[b] Veritas DGC Inc., Calgary, Canada

ABSTRACT

Visual simulation can be efficiently performed using programmable graphics hardware. However, in utilizing hardware to maximize throughput, it is important not to constrain interactivity. We present a method of using the graphics hardware while maintaining full interactivity during simulation exploration. This interactivity involves: temporal exploration, data probing and modification, simulation model modification, and user defined visual metadata. Results are shown using our application for exploring a reaction-diffusion simulation.

Keywords: interactive simulation, visualization, graphics hardware acceleration

1. INTRODUCTION

In this paper we describe how utilizing the graphics processing unit (GPU) makes it possible to include the advantages of 3D temporal and other visualization exploration techniques with visual simulations. GPU techniques have made it possible to create dynamic visual simulations[1,2,3,4,5]. For some of these simulations the visual dynamics are the desired result. For example, creating real-time cloud formation simulations[2] may be exactly what is needed to make a more effective animation. In contrast, for some simulations an important part of the interest may be directed at the process that is being simulated. Examples here include such things as biological, chemical and sociological processes. For these processes it is desirable but not necessarily sufficient to enable animated viewing of the simulation. Direct animation of a simulation does not necessarily provide access to the full richness of information available in the simulation data. For example, it may be the information about when and where crucial changes take place during the process that are of particular interest. In fact, the problem can be that the researcher may not know exactly what he/she needs to see; hence creating a visual simulation that includes dynamic interaction and manipulation at all levels is important to enable full exploration of the available data. We propose the integration of visualization data exploration techniques to support the analysis of the data generated by a simulation. This is achieved at interactive rates through use of the GPU.

In this paper we will describe a general methodology that allows a fully interactive simulation using OpenGL and NVIDIA's Cg language for GPU programming. All major aspects of the simulation and visualization are performed using the graphics hardware (the GPU and VRAM). Our specific implementation will be described as well as a detailed case study of a reaction-diffusion simulation.

2. BACKGROUND

We will briefly review some previous simulation and visualization work using the GPU, and provide motivation for using the graphics hardware to perform simulation and visualization. Harris *et al.*[1] has shown the flexibility of graphics hardware to run coupled map lattice (CML) simulations of various types. A CML is like a cellular autonoma (CA) with real-valued state variables that evolve based on their current state and a set of local neighbours. Li, Wei, and Kaufman[3] demonstrated the use of the GPU to perform fluid simulation using the Lattice Boltzmann Method, and showed up to a 55.9 times speedup over their software implementation. With these and other simulations, the state variables are put into a texture(s), as this data structure is very efficient for graphics processing. Some previous limitations discussed by Harris *et al.*[1] were due to numerical precision, and these have already for the most part been removed with current support for single precision floating-point textures. Another limitation discussed by Harris *et al.* [1] was the slow performance of copying data into a 3D texture with glCopyTexSubImage3D. While this does still seem to be true on particular hardware and drivers, it did not keep us from using 3D textures for their other benefits, and will likely be more

* Further author information: (Send correspondence to T. Z.)
T. Z.: E-mail: zuk@cpsc.ucalgary.ca

Visualization and Data Analysis 2005, edited by Robert F. Erbacher, Jonathan C. Roberts, Matti T. Gröhn, Katy Börner, Proc. of SPIE-IS&T Electronic Imaging, SPIE Vol. 5669 © 2005 SPIE and IS&T · 0277-786X/05/$15

efficient with current generation cards. In later work, Harris *et al.* continued to avoid the OpenGL 3D texture for 3D simulations and instead used a slice tiling scheme within a single larger 2D texture[2].

Visualization has also been performed using the GPU in many ways. Heidrich *et al.*[4] showed the flexibility of the fragment pipeline (SGI pixel texture extension) for line-integral convolution, fog models, and other visualizations. For simulation and visualization, Trendall and Stewart[5] used the GPU to create caustics at interactive rates. Weiskopf, Hopf, and Ertl[6] presented multiple visualizations of dynamic 2D and 3D vector fields using the GPU.

Benefits of using the GPU for a variety of non-graphical uses have recently been demonstrated, including linear algebra[7] and non-linear optimization[8]. The general use of the GPU is also promoted by the organization, General-Purpose Computation Using Graphics Hardware (http://gpgpu.org/). There are various reasons why utilizing graphics card hardware can be beneficial. The advances of graphic card technology and the ease of replacing a graphics card, as compared to a CPU, mean that graphics cards often may be updated on a more frequent cycle. The use of current graphics hardware provides access to parallel processing, matrix and vector operations, and math/graphics functions that may be much faster than on your CPU (e.g. the ATI Radeon X800 has single cycle sin and cos functions). The speed and price benefits of the GPU have even made it a possible replacement to the traditional CPU in cluster design[9].

Current GPU architectures provide two types of graphics pipelines: fragment and vertex. The fragment pipeline is the rasterization pipeline, and as it is optimized for the texturing process it is the better candidate for a grid-based simulation in which each texel holds the variable's state. Programs must be written to run on a specific pipeline, and are called fragment and vertex programs. Fragment programming enables parallel computation while requiring no extra programming constructs. Parallelization will occur based on what the card has available. Current cards such as the ATI Radeon X800 and NVIDIA GeForce 6800 have 16 pixel pipelines, while most computers still usually have one CPU processor (although it may do its own form of limited parallelization). The vertex pipeline, designed for vertex transformation, is flexible enough for generic processing, but is set up for larger granularity (although it may also have multiple pipelines; current cards have up to 8). Kruger and Westermann[7] use the vertex pipeline for sparse matrices and the fragment pipeline for full ones. Thus the value of GPU programming has been shown even when not utilizing the visualization advantages.

GPU programming used to be only available through low-level languages, but now there are higher-level options. NVIDIA introduced their C for Graphics (Cg) language[10], which provides a C level language with various specialized functions for high-level instructions available on the GPU, such as matrix multiplication. One step higher up, a C++ language has been created called Sh[11]. If you are using Microsoft's Direct3D API then it provides its own high level shading language (HLSL). The OpenGL GPU programming language is called the OpenGL Shading Language[12] (GLSlang). For general-purpose programming, a new language called Brook[13] based on a stream programming model also provides support for GPU programming.

3. INTEGRATING VISUALIZATION EXPLORATION TECHNIQUES

Programmable graphics hardware provides a fast and visual simulation environment[1]. Utilizing the GPU to drive a simulation can provide streaming image data with 32 bit floating-point precision, generating a large amount of simulation data. We consider all of this data being generated by the simulation as data to be visualized and explored. Providing a user with the ability to interactively explore a simulation, spatially, temporally, and algorithmically, raises questions about how to manage the system constraints and the large amount of data, and still provide the researcher interactive and flexible tools for visual exploration. We will discuss these interaction and exploration methods and provide one possible solution given current technology.

In order to provide the visual results to the user's screen as fast as possible, it makes sense to avoid the bottleneck between the motherboard and the graphics hardware, and to perform the simulation on the graphics card. We propose performing more visual interaction and analysis on the graphics card as well. Even with the emergence of faster buses to the graphics hardware such as PCI Express, or scan line interleaving (SLI) configurations, generating and processing the data on the graphics card can be more efficient, leaving the CPU free to do other work.

The visual interaction techniques we integrate are: temporal exploration, data probing and modification, simulation model modification and the visualization of metadata. We explain each of these interactions and describe how they can be performed efficiently on the GPU using OpenGL and Cg.

3.1. Flexible Temporal Exploration

Simulation results are usually shown either for the current time of the simulation, or at a given selected time, or sequentially creating an animation. However, to understand the simulation process and how it varies with time, it can be useful to plot the data vs. time. The simulation of a 2D grid of cells with various time varying variables will produce a 3D volume for each variable (for previous example see Carpendale *et al.* [14]). If the simulation is not reversible and a user observes a temporal event and wishes to replay it, then the old results must be stored. In order to avoid the slow archival of data from VRAM to main memory some of the results may be archived in VRAM. This is especially important when the user does not stop the simulation. This archival of results is limited by the amount of VRAM available, but 256MB or more provides considerable flexibility and will likely become commonplace soon. While the simulations usually run on 2D textures, a potentially more useful storage structure is a 3D texture. The previous 2D results can be copied directly into a slice in the 3D texture using glCopyTexSubImage3D.

Using a 3D texture object as a history provides an interface to slice temporally (plotting vs. time) through the results using standard OpenGL 3D texturing. During the simulation process the current state is copied into the texture object's slices in a cyclical manner to avoid moving more than a single slice of texture memory. Thus to texture correctly through the temporal cube, the texture mode GL_TEXTURE_WRAP must be set to GL_REPEAT for the depth/temporal dimension.

3.2. Data Probing and Modification

With the simulation state being maintained in VRAM, it complicates probing and modification of the cell's variables, as this would be more easily done using a standard programming language and the CPU. Limited continuous probing can be done efficiently, as glReadPixels can be used to move data from VRAM to main memory for a few cells (pixels) without much speed impact. If the results are read from the screen buffer, the format and precision are based on the chosen display settings, and the scaling and bias back to the data range may have to be performed.

A user usually does not need to modify the simulation data by hand very frequently so this can be done when the simulation is paused, or may be driven by user input events. For example the user may want to paint some cell's variable values with the mouse, in order to consider an alternative scenario that may be difficult to generate through normal simulation. The overhead of moving the grid state in the floating-point texture to main memory, modifying it, and uploading it will not be noticed if the simulation is paused, and the slowdown will be expected, and tolerated, if driven by transient user input. If enough data needs to be modified on the fly, then additional GPU programs can be written to do the modification directly in VRAM.

3.3. Simulation Model Modification

For complete simulation exploration it may be useful to modify the simulation algorithm during the simulation. With an interpreted language, or the use of Microsoft Visual Studio's "Edit and continue" feature, this is a possibility. These techniques have drawbacks, such as speed and platform limitations. The use of NVIDIA's Cg language provides the flexibility of run-time modification, as it can be loaded and compiled during program execution. The ability to modify and dynamically reload Cg programs allows a researcher to adapt or tune the algorithm in the middle of the simulation run, providing the same data constraints exist. The entire state is maintained in the floating-point textures so new programs can be swapped in at any point to change the simulation model and continue on.

If the simulation model needs to be edited the user must understand the programming language, and few researchers outside of computer science want to use GPU assembly (and few in computer science). Cg's "C"-like syntax is simple enough for research scientists with little GPU programming knowledge (or interest) to easily change the mathematical model.

3.4. Visual Metadata

Often the simulation variables themselves may not provide the visual insights the researcher is looking for. It may be of use to provide for the calculation of specific operators such as gradients, edge-detection, flow, thresholds, or metrics, continuously while the simulation runs. While a limited number of additional operators could be provided, the user may also want to write new or modify existing operators. It is also most efficient to perform the operators that are image-based directly on the GPU. For this purpose we can allow the user to select and modify Cg programs in the middle of a simulation. These programs can operate on the current state, and/or previous state, to generate a metadata image.

4. IMPLEMENTATION

Before discussing our application, we will briefly review the process of mapping a 2D simulation into the GPU fragment pipeline. Next an example of Cg code used in a simulation will be presented to illustrate that part of the translation process, as the mathematical model must be translated. Then our application will be presented describing its general features.

4.1 Simulation Domain Mapping

In order to utilize the GPU to run the simulation in a fragment program, the state variables must exist in a regularly gridded texture. Thus the physical spatial extent (or domain) must be mappable to a 2D grid. If the simulation is not already grid-based, this can be done by isotropic or anisotropic sampling at an appropriate resolution. 3D simulations can be mapped in the same way, but due to the large data requirements may need to be handled differently and are not discussed further in this paper, see Harris *et al.*[1, 2] for examples. Special topological requirements may be encoded using a texture channel, to flag simulation boundaries, or store a height map, but this overhead is encountered on every grid cell update. Non-rectangular regions can be processed efficiently by using a Z-buffer to flag cells for which the simulation should not run. Invalid cells will fail the Z test and the fragment program will not be executed. Potentially three channels can be used to store a 3D spatial position for each cell. If the spatial mapping is too complex and the grid size not too large, the utilization of the vertex pipeline may be more appropriate as it is not designed around the concept of a regular grid. In our implementation we utilized pixel buffers (called pbuffers, OpenGL extension WGL_ARB_pbuffer) to use the fragment pipeline to render into off-screen textures with 32 bit floating-point precision.

4.2 Cg Code Example

In this section we will provide an example of the Cg language. A common operator for simulation is the Laplacian, which we will use later in our example of a reaction-diffusion simulation. The 2D Laplacian of a variable *u* when approximated with the Euler forward method can be computed efficiently on the GPU. For example, the Laplacian of *u* is defined as

$$\nabla^2 u = \frac{1}{\Delta x^2}\left(u_{i-1\,j} + u_{i+1\,j} + u_{ij-1} + u_{ij+1} - 4 \cdot u_{ij}\right), \tag{1}$$

where x is the grid cell size, which we have set to unit length (so the denominator constant is 1). A Cg subroutine for computing this diffusion on a floating-point texture is as follows:

```
float3 diffusion(vf30 In, texobjRECT prevGrid )   // Euler forward method
{
    float2 ij = In.TEX0.xy;  // current texture co-ordinates determine cell
    float3 cur = f3texRECT(prevGrid, ij);
    // get values of neighbouring cells, sample texture at neighbour co-ordinates
    float3 nc1 = f3texRECT(prevGrid, ij + float2(1.0, 0.0) ); // texture co-ords go from 0 to dim for texRECT, not 0-1
    float3 nc2 = f3texRECT(prevGrid, ij + float2(-1.0, 0.0) );
    float3 nc3 = f3texRECT(prevGrid, ij + float2(0.0, 1.0) );
    float3 nc4 = f3texRECT(prevGrid, ij + float2(0.0, -1.0) );
    return (nc1 + nc2 + nc3 + nc4 - 4*cur);
}
```

The function f3texRECT performs the texture fetch to retrieve a cell's variables. As current GPUs have vector instructions, each math operation (additions and multiplication) is applied to each component of the float vector in

parallel. In our example the diffusion rates for three variables are calculated at the same time (in the red, green, and blue channels). Each vector instruction provides parallel computation of up to 4 different variables (i.e. all components of a RGBA texture). However the subroutine may also be parallelized based on the number of fragment pipelines on the graphics card.

4.3 Application

We have developed an application called Sim-studio to provide the interactive flexibility described in the previous section. It is implemented using C++, Qt 3.2.3, OpenGL, and Cg. The application has two windows. The main window lists the initial conditions, the simulation model to use, an operator to perform, parameters, and controls the simulation time increments and display parameters. The other "Data View" window provides a 3D visualization of the temporal data using orthographic projection.

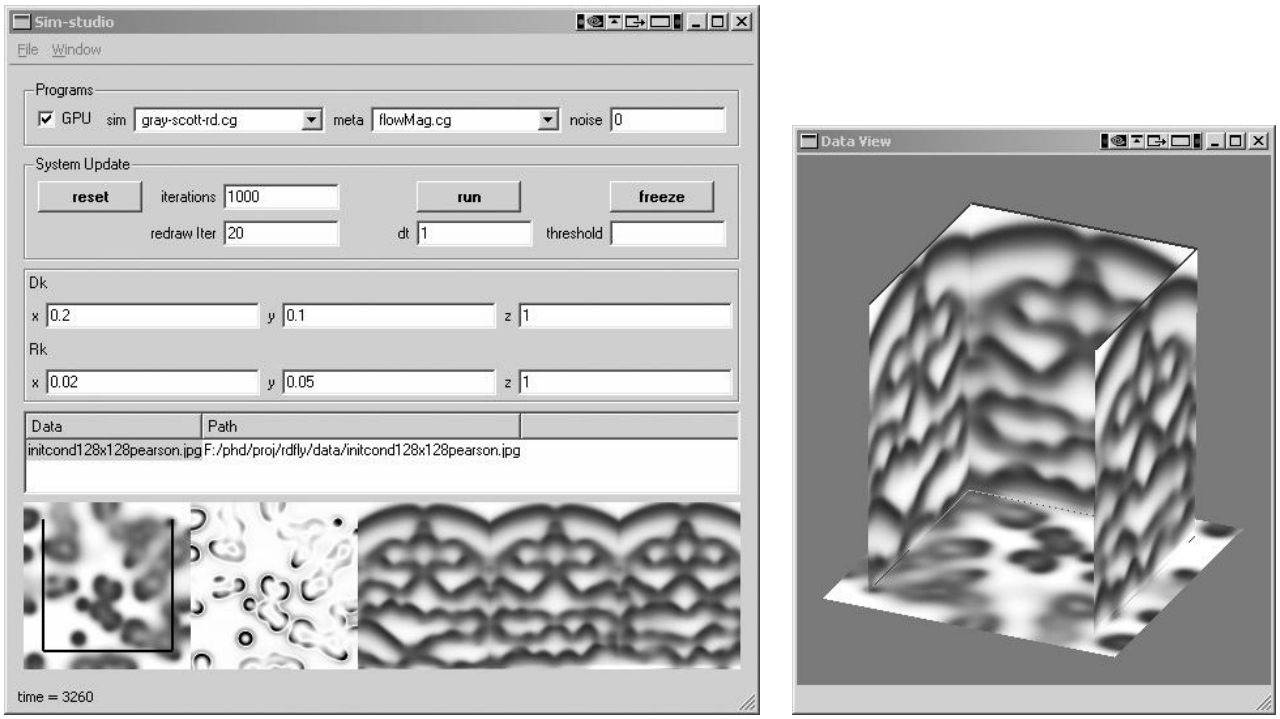

Figure 1. Snapshots of Sim-studio application windows. Shows a reaction-diffusion model as an unstable pattern emerges. The left image shows the main window containing three images at the bottom; from left to right: U concentration (simulation variable), derived data of U diffusion rate, and temporal cuts along black lines from U concentration image (oldest results at top of image, bottom row current simulation state). The right image illustrates the data view window showing the same temporal cutting in 3D.

We will describe the process of running and interacting with a simulation. Initially a program for the type of simulation you wish to run must be selected from the available Cg programs in a drop-down box. A metadata program may also be selected, to determine what derived data should be computed and displayed. Based on the selected simulation, appropriate initial conditions for the grid should then be loaded from an image file, and scaling and bias constants can be specified. Parameters for the simulation program should also be set along with a time step appropriate to the model. The number of iterations to run, and the visual sampling rate (number of iterations per visual update) need to be specified. The visual sampling rate affects not only performance, but also the resolution available for temporal analysis. As the temporal buffer is limited by the 3D texture requirements (VRAM) a lower visual sampling rate also provides for comparison over longer periods.

Before starting to run the simulation, initial temporal cutting planes may be set. Then the simulation is started during which all 2D and 3D views are updated in real-time based on the visual sampling rate. After, or while, the specified number of iterations are run, the user may manipulate the temporal slicing, edit the model and viewing parameters, edit

the simulation code, or edit the metadata calculation code. After modifying any Cg code, the user presses a key to reload all Cg programs. Then the simulation may be continued from its current state, or reset to start again.

5. RESULTS FOR REACTION-DIFFUSION

This section illustrates the use of Sim-studio with a reaction-diffusion simulation. We start by explaining the basics of reaction-diffusion models and then proceed with a sample interaction with a Gray-Scott reaction-diffusion simulation.

5.1 Reaction-Diffusion

Texture generation using reaction-diffusion came to the forefront of computer graphics with Turk[15] and Witkin[16], but has its origins with the early Computer Science work of Turing[17]. Biological modeling using reaction-diffusion also has a long history[18] and continues to be an area of active research[19]. Pearson presented a concise description of the Gray-Scott model of a two-reaction system[20], which we will briefly summarize. Two reactants, *U* and *V*, combine to produce more of the reactant *V*. *V* is also consumed by another reaction to produce *P*. A feed process is also continually affecting the two-reactant concentrations *U* and *V*. The reaction part is expressed in the equations:

$$U + 2V \xrightarrow{k_1} 3V$$
$$V \xrightarrow{k_2} P \quad . \tag{2}$$

It is straightforward to transform these two reactions into differential equations of time. Incorporating only the reaction process provides the two first derivative equations:

$$\frac{dU}{dt} = -k_1 \cdot UV^2$$
$$\frac{dV}{dt} = k_1 \cdot UV^2 - k_2 \cdot V \quad , \tag{3}$$

and moving into two dimensions we require the Laplacian in order to incorporate diffusion

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \quad . \tag{4}$$

After adding the feed process (which removes/adds the reactants based on constant *F*) and normalizing to remove the k_1 and k_2 rate constants, the Gray-Scott partial differential equations for the rate of change in *U* and *V* are

$$\frac{\partial U}{\partial t} = D_u \nabla^2 U - UV^2 + F(1-U)$$
$$\frac{\partial V}{\partial t} = D_v \nabla^2 V + UV^2 - (F+k)V \quad . \tag{5}$$

These equations have been shown to converge to various patterns given the entire grid in the trivial steady state of *U*=1.0 and *V*=0.0, and then a small arbitrary region initially perturbed to *U*=0.5 and *V*=0.25. Results from a 128x128 grid are shown in Figure 2 for a selected set of *F* and *k* parameters. Regions of low *U* and high *V* concentration appear to replicate themselves over time in much the same way as a cellular division. The pairs of images in Figure 2 show nearly converged systems of spots, stripes, and a hybrid pattern.

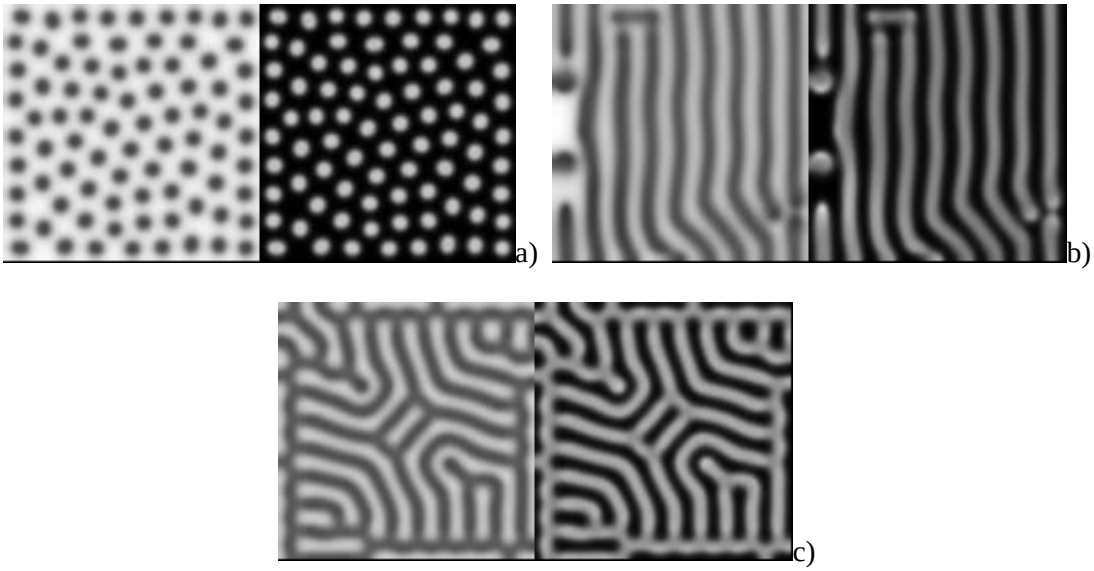

Figure 2. *U* and *V* concentrations after approximately 50000 iterations (*dt*=1) for three different sets of parameters. High concentration is white and low black. a) $F = 0.04$, $k = 0.065$ b) $F = 0.24$, $k = 0.055$ c) F=0.4, k=0.06 (*U* is the left image of each pair). $Du = 0.2$ and $Dv = 0.1$ for all three.

The Gray-Scott reaction is just one possible reaction. Therefore the reaction-diffusion process can be generalized in terms of the two reaction equations

$$\frac{\partial u}{\partial t} = \nabla^2 u + \gamma f(u,v)$$
$$\frac{\partial v}{\partial t} = d\nabla^2 v + \gamma g(u,v) \quad . \tag{6}$$

In these equations if $d > 1$, then *u* is referred to as the activator and *v* the inhibitor. Gamma allows the reaction to be scaled to the same time scale as the diffusion. Functions *f* and *g* must have multiple steady states in order to create patterns. Other constraints with regards to the instability of the steady state are required to create a "Turing System" [19]. These functions *f* and *g*, and the parameters *d* and γ determine what types of patterns can be created.

5.2. Interacting with the Gray-Scott Model Visual Simulation

This section illustrates some possible interactions using our implementation to run a reaction-diffusion simulation of the Gray-Scott model. The interface supports user input for all of the reaction diffusion constants. The initial conditions are set by loading an image file in which the red and green channels indicate the *U* and *V* concentrations, respectively. For this example, the initial conditions are a small square in the centre of the grid set to concentrations U=0.5 and V=0.25, with the rest of the grid set to U=1.0 and V=0.0. The two diffusion rate constants, *F*, and *k* are all specified using widgets and can be changed during the simulation. The time step for each iteration is also specified, with its size being limited by numerical stability of the mathematical model. The current state of the simulation is set to be displayed every 20 iteration/time steps. Figure 3 provides the sequence of 4 images showing output during this simulation. The left most image for each time point shows the *U* concentration as a 2D texture. The red line on these images has been user indicated to select which temporal parts of the simulation should be shown. For the central images, a Cg program has been used to compute the temporal gradient of *U* over each time step and is color-coded red for increasing concentration and blue for decreasing. The right most image series shows the selected temporal cross sections as chosen by the red line in the left image. The bottom of these images is the current time step and the top the oldest stored data from the simulation. Note how for time steps 100 and 600, this right image is only partial because the simulation has not been running for long enough to provide sufficient history data.

Numerous mathematical operators can be performed on the variables by editing a Cg program and the results shown continuously with the simulation (the middle column of images in the Figure 3 sequence). As both the simulation code and the metadata operator code can be modified in the middle of a simulation run, it provides a great deal of flexibility to the user. After editing a Cg program, the reloading and recompiling requires less than a second so the user is provided all the power of an interpreted language. The power of this type of interactivity is often lost with the push to optimize GPU code by hand, while often gaining less than a 50% speedup.

The temporal results cube can have arbitrary cutting planes. These cut planes are continually redrawn with the simulation, both flattened out on a 2D plane, so as to avoid any occlusion, as well as in a 3D view shown in Figure 4. While the simulation is running, the user may create and drag the 3D slice planes arbitrarily to get additional views of the temporal behaviour. These cuts may also be a polyline drawn to specify a shape or follow a feature.

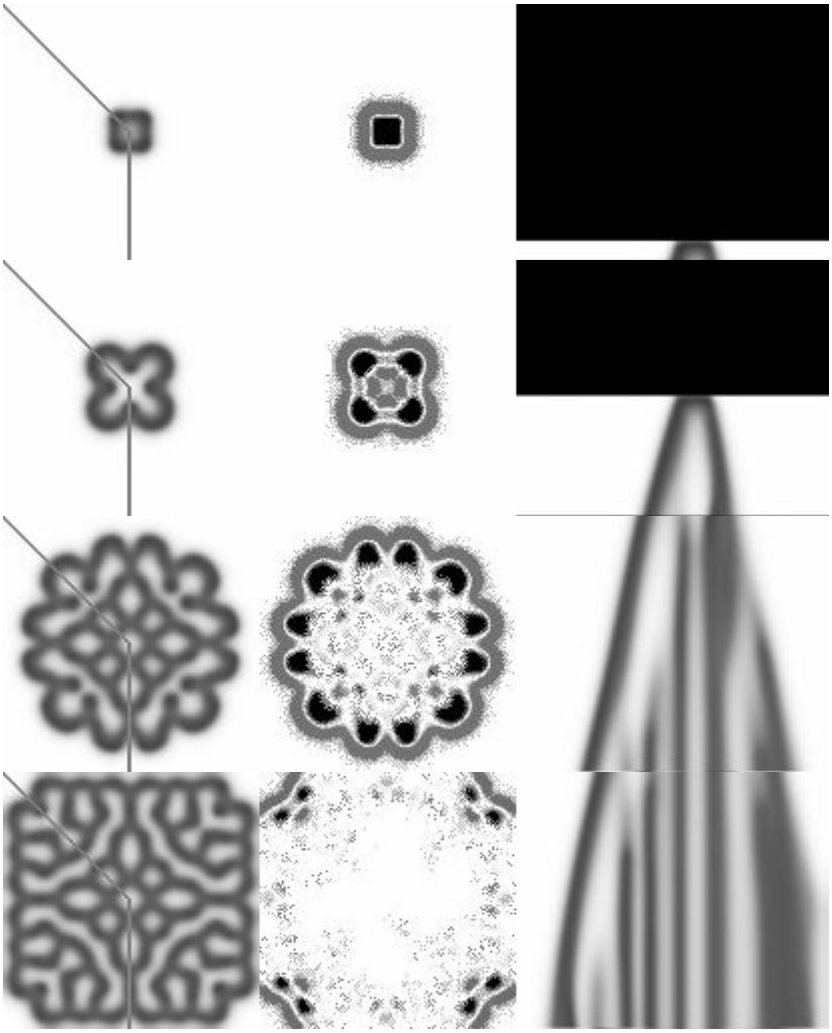

Figure 3. Reaction-diffusion sequence. From left to right: *U* concentration (grey lines show 3D texture cutting planes), *U* concentration temporal gradient (black is positive, grey negative), 3D texture cutting planes with time decreasing vertically (shows 3D temporal buffer filling bottom up). From top to bottom shows iteration 100, 600, 2000, and 3000. *V* concentration is not shown.

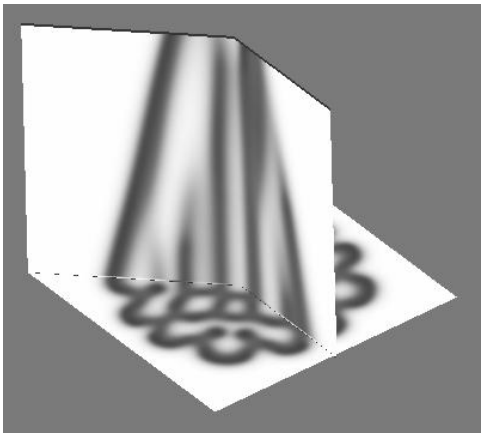

Figure 4. 3D rendering of the temporal slice planes in situ. Shown at iteration 2000. 3D rendering is continually updated during simulation. Time decreases vertically, and the current state is plane at the bottom.

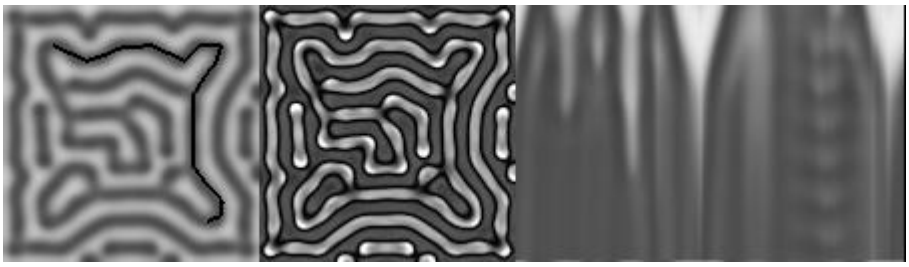

Figure 5. Reaction-diffusion sequence. From left to right: *U* concentration (black lines show 3D texture cutting polyline), *U* diffusion rate (light is positive, dark negative), 3D texture cutting planes flattened out with time decreasing vertically (shows different convergence rates and some oscillatory behaviour).

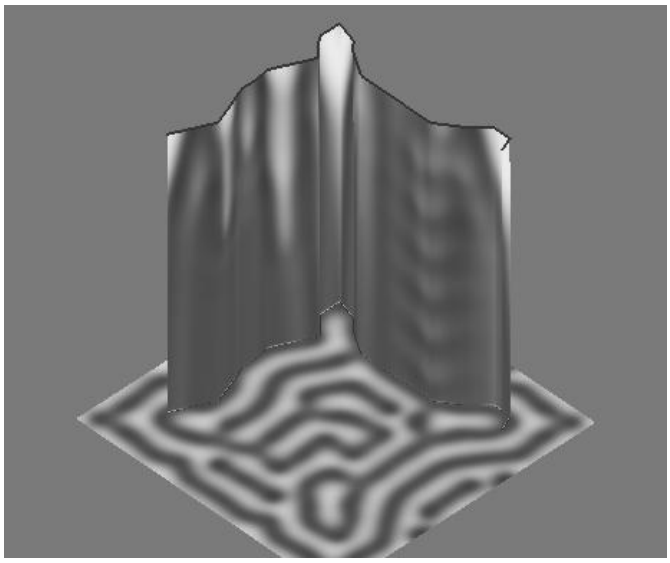

Figure 6. 3D rendering of the temporal polyline slice in situ. Time decreases vertically, and the current state is the plane at the bottom (shows different convergence rates and some oscillatory behaviour).

5.3. Performance with the Gray-Scott Model

To provide some performance numbers Sim-studio was run on two different computers. The first used a NVIDIA GeForce FX 5600 (256MB, AGP 8X, driver 6.14.10.6693) with an AMD XP 2600+ running Windows 2000. The second more current configuration was a NVIDIA GeForce 6800 GT (256MB, PCI-E 16X, driver 6.14.10.6693) with an Intel Pentium 4 HT 3.2 GHz running Windows XP. The AGP/PCI-E difference should be negligible as almost all computation was performed on the card and so the bus is not much of a factor. The results are for the most part CPU independent as well, as on the older configuration changing the AMD XP processor from 1600+ to a 2600+ and increasing the front-side bus speed from 266MHz to 333MHz had almost no impact, showing the throughput was GPU bound. The two configurations will hereafter be referred to by the graphics card only, but are not intended as a graphics card comparison due to the other differing hardware and operating systems. They should be used as indicative of a 1 to 2 years difference in consumer (gamer) hardware.

Simulation throughput is shown in Figure 7, with two graphs of the same data using different representations. They graph the speed of only the simulation for the 6800 GT and 5600 FX configurations across various square grid sizes on a grid iteration and cell update basis. The cell throughput is simply the grid iterations per second times the grid size squared. This figure shows the general iteration rates that can be achieved and reveal the overhead associated with each iteration (OpenGL context switches), as throughput is greatly increased with larger grids. The render to texture OpenGL extension WGL_ARB_render_texture could avoid the context switch overhead as pbuffer targets were swapped, but this is currently only a Windows platform solution and so was not used.

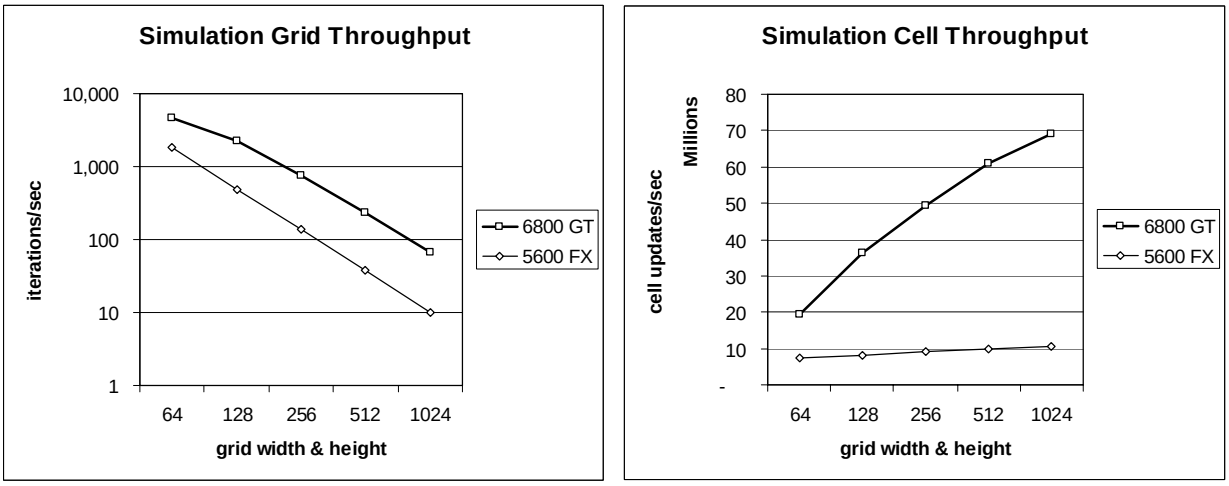

Figure 7. Simulation throughput for various square grid sizes on two different computer configurations.

Interactive simulation visual frame rates on a 128x128 and 256x256 grid are summarized in Table 1. After each ten grid iterations it performs all of the following: gradient calculation, 3D texture updating, and the rendering of one variable, the gradient metadata, and the 3D temporal volume slices in a 2D view and in 3D in an approximately 256x256 window with a separate OpenGL context (shared resources). These numbers show that smooth animation is possible with the full interactivity described previously.

Table 1. Complete visualization feedback in frames per second with 10 iterations between frames.

Interactive Simulation Rates (frames/sec)		
grid size	6800 GT	5600 FX
128	30.3	20.8
256	25.0	8.3

6. CONCLUSIONS AND FUTURE WORK

We have described a methodology and an application that provides a fast and flexible way to interactively explore simulations on the GPU. Often new visualizations are presented as if the only goal is how fast it can be performed, but a fast simulation does not provide much value if the researcher can not explore the data on their own terms. We have shown how some powerful interaction features such as real-time temporal exploration can be provided while utilizing the GPU.

There are some design constraints that may affect the ability to adopt the GPU for many types of simulation. If numerical stability or accuracy requires greater precision (e.g. double) than available on the GPU then you would use have to use some tricks to obtain this, and it would likely negate the benefits of using the GPU. With OpenGL on current graphics cards the maximum texture size is often 2048x2048, or 4096x4096, and therefore grids larger than what your card supports would require some tiling scheme. Total texture memory must also not be exceeded to avoid swapping. For example, with 256MB VRAM approximately 240MB may be available for textures so you should be able to simulate 4 variables (RGBA with 4 bytes per variable) on a 1024x1024 grid with two state buffers (previous/current) using 32 MB, and if the history of one variable is converted to 8 bit for the 3D texture you can have 128 temporal slices using 128 MB. While the use of Cg does not limit you to NVIDIA GPUs, it is only supported on Windows, Linux, and Mac OS X. OpenGL 2.0's inclusion of GLSlang[12] in the specification may in the future make it the definitive cross-platform language. A final major drawback to GPU programming is the lack of direct support for any debugging.

There is future work required in specific domains to determine how researchers utilize these interaction techniques as well as how to simplify adding new visualizations. In the current application, widgets for Cg program parameters are hard-coded. In order to simplify user interaction with modified Cg programs it would be possible to dynamically create widgets based on parameters that are found in the selected Cg program.

ACKNOWLEDGMENTS

Acknowledgement goes to Przemyslaw Prusinkiewic and the University of Calgary Modeling and Visualization of Biological Systems class, for providing motivation for this work. This work has been supported in part by NSERC and Veritas DGC Inc.. NVIDIA Corporation has also supported this work by providing a GeForce 6800 GT graphics card. Thanks also go to Gretchen Greer and Xander Zuk for proofreading and their input.

REFERENCES

1. M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. "Physically-Based Visual Simulation on Graphics Hardware," in *Proceedings of 2002 SIGGRAPH/Eurographics Workshop on Graphics Hardware*. pp. 109-160, 2002.
2. M. J. Harris, W. V. Baxter III, T. Scheuermann, and A. Lastra. "Simulation of Cloud Dynamics on Graphics Hardware," *Graphics Hardware*. 2003.
3. W. Li, X. Wei, and A. Kaufman. "Implementing Lattice Boltzmann Computation on Graphics Hardware," *The Visual Computer*. **19**, No. 7-8, pp. 444-456, 2003.
4. W. Heidrich, R. Westermann, H.-P. Seidel, and T. Ertl. "Applications of Pixel Textures in Visualization and Realistic Image Synthesis," in *Proceedings of ACM Symposium on Interactive 3D Graphics*. 1999.
5. C. Trendall, and A. J. Stewart. "General Calculations using Graphics Hardware, with Applications to Interactive Caustics," in *Proceedings of Eurographics Workshop on Rendering 2000*. Springer, pp. 287- 298, 2000.
6. D. Weiskopf, M. Hopf, and T. Ertl. "Hardware-Accelerated Visualization of Time-Varying 2D and 3D Vector Fields by Texture Advection via Programmable Per-Pixel Operations," in *Proceedings of Vision, Modeling, and Visualization 2001*. pp. 439-446, 2001.
7. J. Kruger, and R. Westermann. "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Transactions on Graphics (TOG)*. **22**, No. 3, (SIGGRAPH '03) pp. 908-916, 2003.
8. K. E. Hillesland, S. Molinov, and R. Grzeszczuk. "Nonlinear optimization framework for image-based modeling on programmable graphics hardware," *ACM Transactions on Graphics (TOG)*. **22**, No. 3, (SIGGRAPH '03) pp. 925-934, 2003.
9. Z. Fan, F. Qui, A. Kaufman, and S. Yoakum-Stover. "GPU Cluster for High Performance Computing," in *Proceedings of the ACM/IEEE Supercomputing Conference 2004*. (To Appear).

10. W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. “Cg: a system for programming graphics hardware in a C-like language,” *ACM Transactions on Graphics (TOG)* **22**, No. 3, (SIGGRAPH ’03) pp. 896-907, 2003.
11. M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. “Shader Algebra,” *ACM Transactions on Graphics (TOG).* **23**, No. 3, (SIGGRAPH ’04) pp. 787-795, 2004.
12. R. Rost. *OpenGL Shading Language*. Addison Wesley Professional. 2004.
13. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. “Brook for GPUs: Stream Computing on Graphics Hardware,” *ACM Transactions on Graphics (TOG).* **23**, No. 3, (SIGGRAPH ’04) pp. 777-786, 2004.
14. M. S. T. Carpendale, D. J. Cowperthwaite, M. Tigges, A. Fall, and F. D. Fracchia. “The Tardis: A Visual Exploration Environment for Landscape Dynamics**,”** In *Proceedings of SPIE, Conference on Visual Data Exploration and Analysis VI*. January, 1999.
15. G. Turk. “Generating Textures on Arbitrary Surfaces Using Reaction-Diffusion,” *Computer Graphics.* **25**, No. 4, (SIGGRAPH ’91) pp. 289-298, 1991.
16. A. Witkin, and M. Kass. “Reaction-Diffusion Textures,” *Computer Graphics.* **25**, No. 4, (SIGGRAPH ’91) pp. 299-308, 1991.
17. A. Turing. “The Chemical Basis of Morphogenesis,” *Philosophical Transactions of the Royal Society*. **237**, pp. 37-72, 1952.
18. A. Gierer, and H. Meinhardt. “A theory of biological pattern formation,” *Kybernetik*. **12,** pp. 30-39, 1972.
19. H. Shoji, Y. Iwasa, and S. Kondo. “Stripes, spots, or reversed spots in two-dimensional Turing systems”. *J. Theor. Biol*. **214,** pp. 549-561. 2002.
20. J. E. Pearson. “Complex Patterns in a Simple System”. *Science* **261**, pp.189-192. 1993.