Procedia Computer Science

Volume 51, 2015, Pages 1986–1995

ICCS 2015 International Conference On Computational Science

TSGL

A Thread Safe Graphics Library For Visualizing Parallelism

Joel C. Adams, Patrick A. Crain, and Mark B. Vander Stel
Department of Computer Science
Calvin College, Grand Rapids, Michigan, USA.
adams@calvin.edu, {pac3,mbv26}@students.calvin.edu

Abstract
Multicore processors are now the standard CPU architecture, and multithreaded parallel programs are needed to take full advantage of such CPUs. New tools are needed to help students learn how to design and build such parallel programs. In this paper, we present the *thread-safe graphics library* (TSGL), a new C++11 library that allows different threads to draw to a shared *Canvas*, which is updated in approximate real-time. Using TSGL, instructors and students can create visualizations that illustrate multithreaded behavior. We present three multithreaded applications that illustrate the use of TSGL to help students see and understand how an application is using parallelism to speed up its computation.

Keywords: Actors, computational science, design patterns, education, graphics, image processing, integration, multithreading, parallel, thread, visualization

1 Introduction

Prior to 2006, most CPUs had a single computational core that could perform one statement at a time. Since 2006, manufacturers have exploited Moore's Law to produce CPUs with multiple computational cores. For example, one may today purchase Intel Xeon CPUs with 2, 4, 6, 8, 10, 12, or 15 cores [10]. Each core in such a CPU can do as much computational work as a single core CPU. Put differently, today's CPUs are multiprocessors that can run multiple programs in parallel.

The statements in traditional software programs are executed one at a time, sequentially. Such sequential programs are said to have a single *thread* of execution. These programs are (by themselves) unable to take advantage of the capabilities of a multicore CPU, since the program's one thread will be scheduled on one of the CPU's cores, while the other cores do nothing. A single-threaded program thus squanders 75% of the hardware capabilities of a modest quad-core CPU.

However, most modern languages can divide a computation among multiple threads of execution. The different threads in such *multithreaded programs* can be scheduled on the different cores of a multicore CPU, so that they run in parallel. For a problem that can be divided among the threads, this parallel execution allows the problem to be solved faster than is possible otherwise.

doi:10.1016/j.procs.2015.05.463

To take full advantage of these multicore CPUs, programs must thus be designed and implemented as parallel programs, and multithreading is the most common mechanism for writing parallel programs for multicore CPUs. This creates a challenge for computing educators, who must now begin creating educational materials and tools to help their students understand parallel computing, multithreading, and a host of related topics (e.g., computational speedup, efficiency, scalability, etc.).

In this paper, we present a new tool called the *thread safe graphics library* (TSGL). As its name implies, this tool provides a library by which multiple threads can safely perform graphical operations. Using it, different threads can simultaneously draw to the screen, in approximate real-time.

Using TSGL, a computing educator can take an application and annotate it with graphics calls so that a thread draws to the screen as it computes, and thus visually displays its contribution to the overall computation. When a single thread performs the computation, one sees that thread producing its results sequentially. When the computation is divided among multiple threads, one sees each of the threads producing their results in parallel, in approximate real-time. Such an annotated application can help one's students see exactly how the application is using parallelism to solve the overall problem. In short, TSGL is a tool designed to make it relatively easy for computing educators and students to build visualizations of multithreaded behavior, to better understand how parallel computations work.

1.1 Objectives

If multiple threads can simultaneously call a function/method without producing a race condition, even when the calls will access shared data, then the function/method is called *thread-safe*. In the rest of this paper, we will use the word *safe* as synonymous with thread-safe.

With this in mind, our list of objectives for TSGL included:

- A *Canvas* class for 2D graphics, to which multiple threads can safely draw (or read from).
- A *Shape* class hierarchy for drawing shapes such as lines, polygons, circles, and so on.
- A *CartesianCanvas* class (a subclass of *Canvas*) for safely and conveniently creating graphical Cartesian coordinate systems, drawing on them, and so on.
- A *Function* class hierarchy for easily plotting functions.
- Support for having and using multiple *Canvas* or *CartesianCanvas* objects simultaneously.
- Support for reading, writing, displaying, and processing PNG, JPEG, and BMP image files; plus safely getting and/or setting the individual pixels in such images.
- Support for interacting with a *Canvas* using a mouse or keyboard.
- Platform independence and high performance.
- Various *Color* classes, plus methods by which a thread may get and draw with a unique color, so that the items drawn by different threads may be easily distinguished from one another.
- A *Timer* class by which thread-executions can be easily delayed, to make it easy to slow down a computation when desired.
- Operability with C++11, OpenMP, and POSIX threading.
- HTML-based API documentation similar to the Java API.

In a nutshell, our goal is that when a computation creates a TSGL *Canvas* object, that action will open a new window on the user's screen. Any of the computation's threads that share that *Canvas* may then safely invoke its methods to draw shapes on it, load images into it, read pixels from it, or write pixels to it; and any effects of those methods will be rendered on the *Canvas* "immediately."

1.2 Related Work

Others have built graphics libraries for CS education to help students visualize program behavior. One of the earliest was a mid-1990s C library from Stanford [9]. More recent efforts (e.g., [2] and [10]) have focused on Java, which is not commonly used for parallel / high performance computing. None of these libraries guarantees thread-safety or C++, two important objectives of our project.

We also examined other C++ graphics libraries, including:

- *Qt* [3]. Qt provides a rich C++ graphical class hierarchy. Unfortunately, it is not thread-safe [4].
- The *Fast Light Toolkit* (FLTK) [13]. FLTK supports many of our objectives, and can be configured to be thread-safe, but it is not thread-safe by default. In the end, we decided against using it because one could not easily take an existing multithreaded application and simply annotate it with FLTK calls to create a visualization of that application's parallel behavior.
- The *MPE* library [15]. MPE permits multiple C or C++ *processes* to safely perform graphical operations in parallel; but MPE's graphics library is not thread-safe as of this writing.

Since none of these existing libraries was fully satisfactory for realizing our objectives, we were motivated to build TSGL.

In Section 2, we describe our design of TSGL and the implementation of that design. In Section 3, we present some examples that demonstrate its capabilities. In Section 4, we conclude with a discussion of how TSGL might be used to help students understand parallel computing concepts

2 TSGL

2.1 Design

In Section 1.1, we presented a list of our initial TSGL objectives. To help achieve those objectives, we designed the class hierarchy shown in Figure 1 to relate the various objects of our objectives.

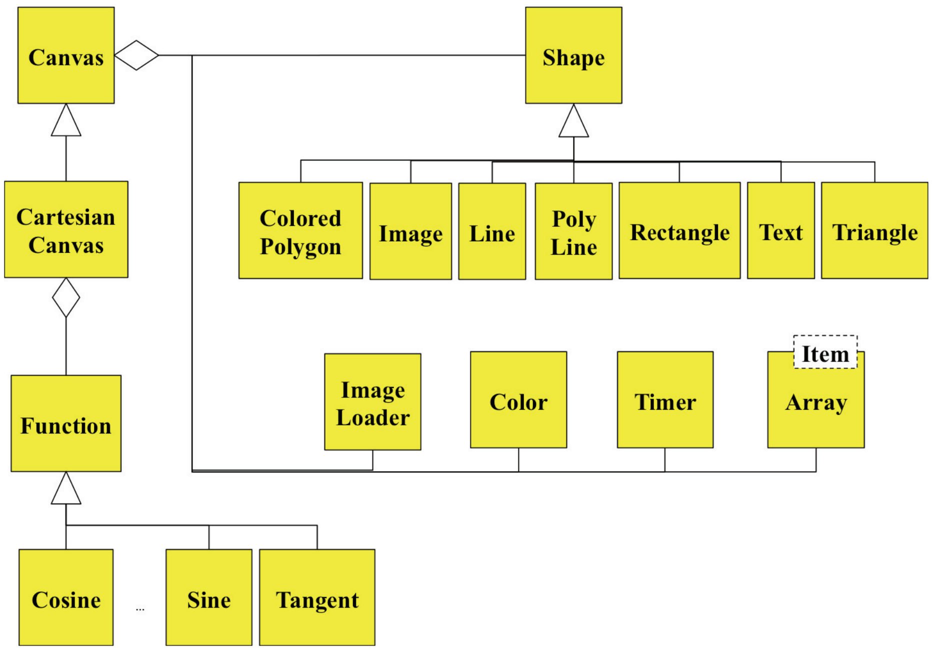

Figure 1. TSGL Class Diagram

The *Canvas* class in Figure 1 plays a central role in TSGL, as it provides the methods to draw the various shapes in the *Shape* class hierarchy. The *Color* class provides methods by which different threads may get unique and contrasting colors.

The *Canvas* class also provides methods for loading, storing, displaying, and manipulating images in the bitmap (BMP), joint photographic experts group (JPEG), or portable network graphics (PNG) formats. These methods rely heavily on the *Image* and *ImageLoader* classes of Figure 1, which provide most of the functionality for handling the different image formats.

Canvas also serves as the superclass for our *CartesianCanvas* class, which makes it easy to create a window containing a Cartesian coordinate system. The *CartesianCanvas* class provides methods for conveniently displaying the x-y axes of a Cartesian coordinate system, transparently mapping (x,y) coordinates in that system to the corresponding pixel coordinates, and plotting *Function* objects. The *Function* class hierarchy defines subclasses for a variety of commonly used functions, and allows other functions to be easily plotted if they are implemented as subclasses of *Function*.

Given modern processor speeds, many graphical computations occur too fast for a person to see essential details, especially when we want to visualize individual thread behaviors. To allow an educator to control a computation's speed, TSGL's *Timer* class provides methods to delay a thread's execution, so that fine details of that thread's parallel behavior can be observed.

2.2 Implementation

To implement our design, we used:

- C++11 as our programming language to ensure that TSGL would be operable with C++11, OpenMP, and POSIX threads. This made it straightforward to build the classes shown in Figure 1, which we compiled with GNU's g++ compiler, version 4.9.
- OpenGL as the graphical foundation for our *Canvas* class, to ensure platform independence and high performance. For handling OpenGL extensions conveniently, we used the OpenGL Extensions Wrangler (GLEW) library [5]. For interacting with the *Canvas* using a mouse or keyboard, we used the GLFW library [11].
- The C++11 *std::chrono* library to build our *Timer* class. This new-in-C++11 library provides support for durations, clocks, time points, and a variety of utility functions, as well as support for C-style dates and times.
- The *libjpeg* [7] and *libpng* [12] libraries, to handle the image data of JPEG and PNG files, respectively. We were able to handle the data from BMP files without external libraries.
- The Doxygen documentation generator [14] to document our library's API in HTML format.

If the same thread performs both computation and graphical rendering, then the computation part of that thread will be unable to execute while the graphical content is being rendered. To avoid this problem, we gave each *Canvas* its own "render-thread" that handles the rendering of the graphical content being drawn, independently of the threads performing the computation.

To allow a computation thread to safely communicate with the render-thread, we used the *shared queue* parallel design pattern. More precisely, when a computation thread invokes a *Canvas* method to draw a *Shape*, that method adds the *Shape* to be drawn to the *Array* member of our *Canvas*. The *Canvas*' render-thread repeatedly retrieves *Shape*s from that *Array*, deposits them in its own buffer, and draws the contents of its buffer on the *Canvas*.

3 Examples

TSGL also includes a tutorial and a variety of examples that illustrate different ways it can be used. In this section, we present three of these examples that can be used to teach parallel design patterns to students of varying experience levels.

3.1 Integration (Area Beneath The Curve)

For students who have had integral calculus, integration should be a familiar concept, and they may be interested in learning how integration can be performed computationally. While there are a variety of methods that can be used, one that is easy for students to understand is to compute the area between the function's graph and the x-axis for the specified range of x values. A pseudo-code version of the algorithm to compute the integral of **f(x)** over the range **a** to **b** might be given as follows:

```
area = 0.0;
width = b - a;
deltaX = width / NUM_RECTANGLES;
startX = a + deltaX / 2;
for (x = startX; x < b; x += deltaX) {
   y = f(x);
   area += y;
}
area *= deltaX;
return area;
```

For each point along the curve, the algorithm's loop uses the variable **area** to accumulate the sum of the y-value "height" distances from the x-axis to the curve. When the loop is completed, we multiply the accumulated "height" distances by the **width** to finish computing the **area**.

Since each iteration of the loop is independent of the others, we can parallelize this computation using the *parallel loop* pattern, in which a loop's iterations are spread across different threads. Given a loop with the following form:

```
for (i = 0; i < MAX; i++) {
   // perform work using index i
}
```

This sequential loop can be transformed into a scalable parallel loop using *parallel loop pattern*:

```
numThreads = getNumberOfThreads();
blockSize = MAX / numThreads;
threadID = getThreadID();
start = threadID * blockSize;
stop = start + blockSize;
for (i = start; i < stop; i++) {
   // perform work using index i
}
```

When one thread executes this code, it will perform all iterations of the loop. However, when two threads execute this code on a computer with two or more cores, thread 0 will perform the first half of the loop's iterations and thread 1 will simultaneously perform the second half of the iterations. When four threads execute the code on a computer with four or more cores, threads 0 will perform the first quarter of the iterations, thread 1 will simultaneously perform the second quarter, thread 2 will simultaneously perform the third quarter, and thread 3 will simultaneously perform the final quarter.

The *parallel loop* pattern thus takes a loop's workload and divides it among the threads. (To keep this presentation simple, we have assumed that the number of iterations **MAX** is evenly divisible by the number of threads. Removing this assumption is not difficult.) Many mathematical computations spend the majority of their execution time performing such for loops, so the *parallel loop* pattern is an important construct for student software developers to master.

Explaining this pattern verbally to students can be challenging, but TSGL makes it possible to explain the pattern visually: In our integration example, we can use annotate the algorithm with calls that (1) create a *CartesianCanvas* object, (2) tell it to display its axes, (3) draw function **f()** on it, (4) get a unique *Color* for our thread, and then (5) for each value of **x**, as the thread computes **y = f(x)**, draw a line of that *Color* from the **(x,0)** to **(x,y)**. The pseudo-code shown below shows this annotation:

```
area = 0.0;
width = b - a;
deltaX = width / NUM_RECTANGLES;
numThreads = getNumberOfThreads();
blockSize = width / numThreads;
threadID = getThreadID();
startX = threadID * blockSize + deltaX/2;
stop = startX + blockSize;
CartesianCanvas canvas(MAX_X, MAX_Y);
canvas.showAxes();
canvas.drawFunction(f);
Color color = canvas.getColor(threadID);
for (x = startX; x < stop; x+=deltaX) {
   y = f(x);
   canvas.drawLine(x, 0, x, y, color);
   area += y;
}
area *= deltaX;
return area;
```

Figure 2 shows the *CartesianCanvas* this cosine-integral computation creates, and how it differs when using one thread vs. using eight threads:

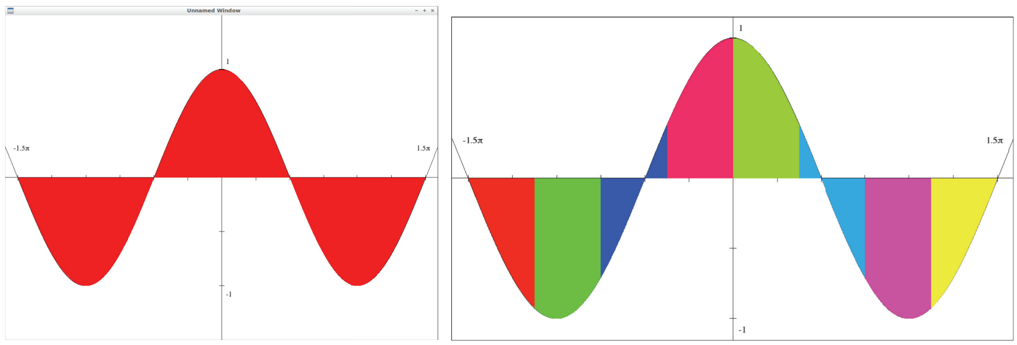

Figure 2. Cosine Integration: One Thread (left) vs. Eight Threads (right)

The non-white regions in each image of Figure 2 indicate the areas the different threads have processed. With one thread (shown in the left image), thread 0 receives the color red and does all the work, so the entire area between the graph and the x-axis is shaded red. With eight threads (shown in the right image), thread 0 has again acquired the color red, but thread 1 has acquired a bright green, thread 2 has acquired a royal blue, thread 3 has acquired a crimson red, thread 4 has acquired a lime green, thread 5 has acquired a sky blue, thread 6 has acquired a magenta, and thread 7 has acquired a bright yellow. As each thread computes its part of the integral, it colors the corresponding part of the graph. By letting a student see exactly what work each thread is doing, TSGL makes it easier for an instructor to build examples like this that can help students understand how a parallel algorithm is dividing the work among the different threads.

Moreover, TSGL does this in approximate real time. If computing the integral with one thread takes time *N*, two threads compute it in about time *N/2*, four threads in about time *N/4*, and so on. On this and other embarrassingly parallel problems, TSGL lets a student *see* the processing time decrease as the number of threads is increased (until the number of threads exceeds the number of cores), providing a visual experience through which an instructor can introduce students to the concepts of speedup, scalability, and so on.

3.2 Image Processing

Integral calculus may be interesting to students who have had calculus, but it may not be very motivating for many first-semester college students. For students who have spent their teen years taking pictures with smart phones, applying a transformation to a large digital image may be a more motivating example. Once they have been introduced in this familiar setting, the same techniques can then be applied to images from scientific domains such as astronomy, medicine, and so on.

In the rest of this subsection, we will discuss the *color inversion* (also known as creating a *negative*) transformation, and will use TSGL to transform a large PNG image. The TSGL *Canvas* class provides a `loadImage()` method that makes it trivial to display such images.

There are different algorithms for inverting a color image, depending on how the image's color information is stored. For images where a color's components are RGB integers between 0 and 255, a pseudo-code version of the algorithm is as follows:

```
Canvas canvas;
canvas.loadImage(imageFile);

for each y in canvas.getRows() {
   for each x in canvas.getColumns() {
      Pixel p = canvas.getPixel(x,y);
      newR = 255 - p.getR();
      newG = 255 - p.getG();
      newB = 255 = p.getB();
      canvas.setPixel(x, y, newR, newG, newB);
   }
}
```

For smaller images, a single thread using this algorithm may be able to invert the image "instantly" – too fast for the human eye to see. For those images, TSGL's *Timer* class can be used to slow that thread sufficiently for students to see it processing the image sequentially.

In this algorithm, each of the loop's iterations are independent of the others, so we can once again use the *parallel loop* pattern to parallelize this algorithm. In Figure 3, the left image shows a colorful picture displayed using a TSGL *Canvas*. The middle image shows a second TSGL *Canvas* in which four threads are using the *parallel loop* pattern to invert the image, when the computation is about two thirds finished. The right image shows that same *Canvas* when the transformation has completed.

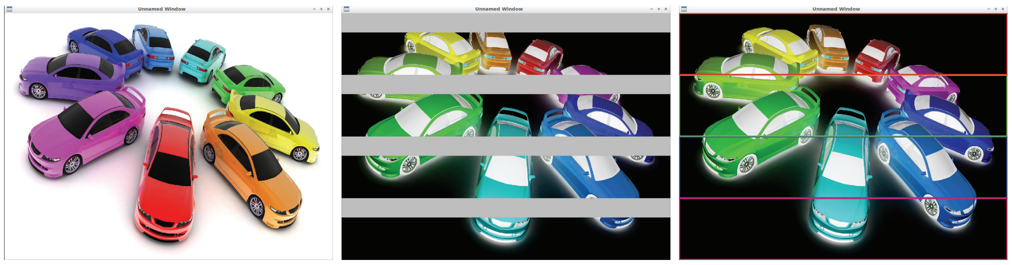

Figure 3. Color Inversion: Original (left), In Progress (middle), and Completed (right)

In the middle image of Figure 3, the four gray bands are the portions of the image on which the threads are still working and the other four bands are the portions the threads have completed. The right image shows how each thread, after it has finished its part of the inversion, gets a unique color and draws a bounding box around its portion of the image, so that a student can easily see exactly what part of the image each thread inverted. TSGL thus lets us write multithreaded applications that allow students to see: (i) *when* a thread is doing its work, relative to the other threads, and (ii) *what work* each thread is contributing to the overall problem solution.

3.3 Actors

Both of the previous examples have used the *parallel loop* pattern. TSGL can also be used to create visualizations of other parallel patterns. For example, in the *actor* pattern, an actor is an autonomous software agent that can repeatedly (i) sense its environment, (ii) change its environment, (iii) receive messages from other actors, (iv) send messages to other actors, (v) make decisions, and/or (vi) create new actors, as necessary. An actor generally has its own thread of execution to provide its autonomy, and that thread contains a loop to drive its repeated behavior. Communication between actors is asynchronous, and an actor can perform behaviors in parallel.

The *actor* pattern can be used to model many real-world phenomena by creating an actor for each real-world entity and programming it with the necessary behavior. An interesting example that can be used to illustrate this pattern is *Langton's ant* [8], in which an actor "ant" moves about a 2D torus of (initially neutral) squares and leaves a trail of colored squares by following two simple rules:

- At a neutral square, turn 90° right, flip the color of the square, move forward one unit
- At a square of the ant's color, turn 90° left, flip the color of the square, move forward one unit

These simple rules produce interesting emergent behaviors. For the first few hundred moves, an ant leaves a trail of simple localized patterns that are often symmetric. Then for about 10,000 moves, the ant's trail forms an irregular jumble. After that, the ant begins following a recurring sequence of 104 moves that creates a linear trail commonly known as "the highway".

The rules are easy to adapt for multiple ant-actors. Each ant is initialized with a *CartesianCanvas* to which it will draw, a starting (x,y) position on the *CartesianCanvas*, a direction, and a unique color. Its behavior is controlled by a loop like that shown in pseudocode below:

```
while (true) {
    Color color = canvas.getPixel(x,y).getColor();
    Color newColor = null;
    if (color == backgroundColor) {
        newColor = myColor;
        myDirection += 90;
    } else {
        newColor = backgroundColor;
        myDirection -= 90;
    }
    canvas.setPixel(x, y, newColor);
    moveForward(myDirection);
}
```

The iterations of this loop are not independent, so the *parallel loop* pattern cannot be used to parallelize it. However the behaviors of one ant are independent of the behaviors of another ant, so the actor pattern can be used to have each ant perform this behavior in parallel.

Figure 4 shows screen captures in which four actor-ants are leaving trails on a TSGL *CartesianCanvas* in which the origin is at the center of the screen but the axes are hidden. Each ant is initially positioned equidistant from the origin, one due north (blue), one due south (magenta), one due east (green), and one due west (red). The direction of each ant was likewise set so as to create a cycle, in which the northern ant builds its highway southwest toward the western ant, the western ant builds its highway southeast toward the southern ant, the southern ant builds its highway northeast toward the eastern ant, and the eastern ant builds its highway northwest toward the northern ant.

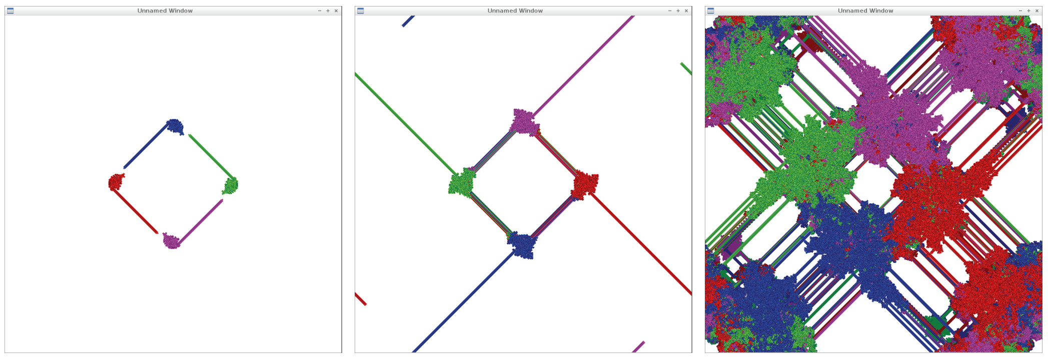

Figure 4. Four Langton's Ants, at Different Stages of Execution

The left image in Figure 4 was captured early in the program's execution, when the four ants are building their initial highways. When an ant encounters the trail left by another ant, it treats that trail as if it were its own trail. This temporarily terminates the ant's highway-building behavior, but it eventually begins building another highway, perpendicular to its original highway, as can be seen in the middle image of Figure 4. The right image in Figure 4 was captured after the program had been running for several minutes, and each of the ants had encountered the others' trails many times.

TGSL includes methods to detect key presses and mouse clicks, which provides support for interactive control. For example, in the program in Figure 4, the *enter* key has been bound to a method that toggles the render thread between running and sleeping, and the *spacebar* has been bound to a method that clears the window's *CartesianCanvas*.

TSGL thus permits actors/agents to create visual representations of the actions they are taking, making it relatively easy for students to see and understand their behavior. When applied to real-world phenomena, such visualizations can provide new insights into those phenomena.

4 Conclusions

TSGL is an application of the old proverb, *"A picture is worth 1000 words."* We believe that one of the most effective ways to help students understand abstract concepts is to create interactive, visual programs that help them see a graphical representation of that concept. With multicore CPUs being ubiquitous, we also believe that every computing student needs to learn how to write software that can take advantage of all of a CPU's cores. With TSGL, we have created a tool that combines these beliefs: a tool that we and other educators can use to create programs that help our students visualize a multithreaded program's parallel behavior in approximate real time.

Work on TSGL began in June 2014, so it is a still-evolving tool. We plan to use it for the first time in the classroom during the Spring 2015 semester. We believe that it holds great potential for creating visualizations of multithreaded behavior, and thus helping students understand parallel computing and employ it in their own projects.

A conference presentation of this paper will include live demos of the visualizations from Section 3, along with others. It is important that such visualizations be scalable by allowing the user to easily change the number of threads being used; the visualization's graphical behavior should change appropriately with different numbers of threads.

We have published TSGL release 1.0 on GitHub [1] as free, open-source software, under the GNU Public License (v.3). It was developed on Linux and has been successfully tested on Windows; a MacOS port is in progress. We invite others to use it and/or contribute to the project, and we look forward to seeing the visualizations that others create with it.

5 Acknowledgements

TSGL was created with support from NSF-DUE #1225739. The authors also wish to thank the reviewers, several of whom had helpful suggestions.

References

[1] J. Adams, P. Crain, M. Vander Stel. TSGL. Online: https://github.com/Calvin-CS/TSGL Accessed 2014-12-12.

[2] K. Bruce, A. Danyluk, T Murtagh. 2010. Introducing Concurrency in CS1. *Proc. 41st SIGCSE Tech. Symp. on Computer Science Education* (Mar. 2010), 224-228. DOI=10.1145/1734263.1734341.

[3] Digia pic. 2014. Qt Project. Online: http://qt-project.org. Accessed 2014-12-12.

[4] Digia pic. 2014. Qt reentrancy and thread safety. Online: http://qt-project.org/doc/qt-4.8/threads-reentrancy.html. Accessed 2014-12-12.

[5] M. Ikits and M. Magallon. The OpenGL Extensions Wrangler Library. Online: http://glew.sourceforge.net. Accessed 2014-12-12.

[6] Intel, Intel Xeon Processor E7 v2 Family. Accessed 2014-12-12. Online: http://ark.intel.com/products/family/78584/Intel-Xeon-Processor-E7-v2-Family#@Server.

[7] T. Lane. libjpeg. Online: http://libjpeg.sourceforge.net. Accessed 2014-12-12.

[8] C. Langton. "Studying artificial life with cellular automata". Physica D: Nonlinear Phenomena (1986) 22 (1-3), 120–149. DOI=10.1016/0167-2789(86)90237-X.

[9] E. Roberts. 1995. A C-based Graphics Library for CS1. *Proc. 26th SIGCSE Tech. Symp. on Computer Science Education* (Mar. 1995), 163-167. DOI=10.1145/199688.199767.

[10] E. Roberts, K. Bruce, J. Cross II, R. Cutler, S. Grissom, K. Klee, S. Rodger, F. Trees, I. Utting, F. Yellin. 2006. The ACM Java Task Force: Final Report. *Proc. 37th SIGCSE Tech. Symp. on Computer Science Education* (Mar. 2006), 131-132. DOI=10.1145/1121341.1121384.

[11] R. Salminen, et al. GLFW. Online: http://www.glfw.org. Accessed 2014-12-12.

[12] G. Shalnat, A. Dilgar, J. Bowler, et al. libpng. Online: http://www.libpng.org/. Accessed 2014-12-12.

[13] B. Spitzak, et al. 2012. Fast, Light Toolkit (FLTK). Online: http://www.fltk.org/. Accessed 2014-12-12.

[14] D. van Heesch. Doxygen. Online: http://www.doxygen.org. Accessed 2014-12-12.

[15] O. Zaki, E. Lusk, W. Gropp, D. Swider. 1999. Toward Scalable Performance Visualization with Jumpshot. *International Journal of HPC Applications*, (13)3, August 1999, 277-288.