Seeing Multithreaded Behavior Using TSGL

Joel C. Adams
Dept. of Computer Science
Calvin College
Grand Rapids, MI USA
adams@calvin.edu

Patrick A. Crain
Dept. of Computer Science
Calvin College
Grand Rapids, MI USA
patrickacrain@gmail.com

Christopher P. Dilley
Dept. of Computer Science
Calvin College
Grand Rapids, MI USA
cpd5@students.calvin.edu

***Abstract*—Since multicore processors are now the architectural standard and parallel computing is in the core CS curriculum, educators must create pedagogical materials and tools to help students master parallel abstractions and concepts. This paper describes the *thread safe graphics library* (TSGL), a tool by which an educator can add graphics calls to a working multithreaded program in order to make visible the underlying parallel behavior. Using TSGL, an instructor (or student) can create parallel visualizations that clearly show the parallel patterns or techniques that a given program is using, allowing students to see the parallel behavior in near real-time as the program is running. TSGL includes many examples that illustrate its use; this paper presents a representative sample that can be used, either in a lecture or a self-paced lab format.**

Computer science; education; graphics; library; multicore; multithreading; parallel; thread safe; threads; visualization.

I. INTRODUCTION

The vast majority of central processing units (CPUs) being manufactured today have multiple cores, and each of a CPU's core can execute different statements in parallel in real-time. Quad-core CPUs are common, and with sufficient budget, one can purchase CPUs with 8, 12, 16, and even more cores.

Traditional sequential programs will not run faster on such CPUs, as such programs have a single thread of execution. Indeed, as such programs are run on CPUs with more and more cores, sequential programs use the available hardware less and less efficiently, as illustrated in Figure 1:

Available cores	Sequential Program Hardware Utilization %
1	100
2	50
4	25
8	12.5
16	6.25

Figure 1. Sequential Program Hardware Utilization %

To take advantage of multicore hardware, programs must be designed and written as *parallel programs*, with multiple threads of execution. An effective multithreaded program not only uses the underlying hardware efficiently, it also runs faster on such hardware. Put differently, its performance *scales* with the number of available cores.

Since multicore CPUs are the hardware foundation on which virtually all of today's software will run, it follows that future software developers need to learn about parallel programming in general and multithreaded programming in particular. Accordingly, where parallel computing used to be an elective topic, it is now a core topic in both the *IEEE TCPP Curriculum Recommendations* [4] and the *ACM/IEEE CS 2013 Curriculum Recommendations* [5].

Most future software developers are trained by computer science (CS) faculty members. It is thus the responsibility of CS faculty members to ensure that the students they train learn about parallelism. That is, CS faculty members must create and use tools and pedagogical materials that will help their students understand parallel abstractions and concepts.

There is an old saying:

"A picture is worth 1000 words."

This saying claims that one well-done graphical presentation of information can communicate as effectively as a lengthy textual or verbal presentation. Believing this saying to be true (especially for visual learners), we began trying to create real-time visualizations of multithreaded behavior.

More precisely, we began searching for a graphics library that was *thread-safe*, meaning it would allow multiple threads to write to the screen without producing a race condition. Imagine our surprise when we were unable to find one anywhere! We examined nearly a dozen graphics libraries, and none of them would guarantee thread-safety.

The problem is that graphics libraries define a data structure called a *frame buffer*, to store the graphical information being displayed on the screen. This frame buffer resides in memory, and if two threads try to write graphical information to it at the same time, a data race occurs, usually causing the multithreaded program to crash.

Unable to find a graphics library that was thread-safe, we decided to create one. We (rather unimaginatively) named our creation the *thread-safe graphics library*, or TSGL.

In Section II, we provide an overview of TSGL, and Section III presents several examples that illustrate how it can be used to let students see multithreaded in near real-time. We conclude in Section IV with a discussion of the principles we use in creating such visualizations.

II. TSGL

In this section, we present our design goals for TSGL, our design, and some of the implementation details.

A. TSGL Design Goals

Our list of objectives for TSGL included:

- An easy-to-use *Canvas* class supporting 2D graphics, to which multiple threads can safely draw (or read) pixels.
- A *Shape* class hierarchy for drawing basic shapes such as triangles, rectangles, circles, polygons, and so on.
- A thread-safe *CartesianCanvas* class (a subclass of *Canvas*) to easily make Cartesian coordinate systems.
- A *Function* class hierarchy for easily plotting functions.
- The ability to create and display multiple *Canvas* or *CartesianCanvas* objects, simultaneously or in sequence.
- 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.
- Interacting with a *Canvas* using a mouse or keyboard.
- Support for each thread to draw in a unique color, so that items drawn by different threads can be easily identified.
- Support for easily delaying a thread's execution, if slowing down a computation is desired.
- Platform independence and high performance.
- Operability with C++11, OpenMP, and POSIX threads.
- HTML-based API documentation like the Java API.

B. TSGL Design

To achieve our design goals, we designed classes to provide the needed functionality and organized them into a class hierarchy, part of which is shown in Figure 2:

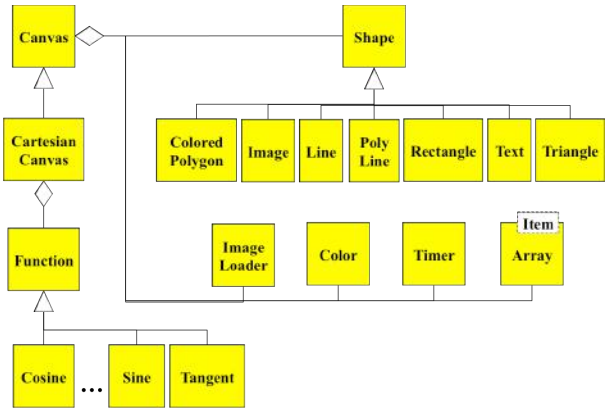

Figure 2. A Partial TSGL Class Structure Diagram

For example, the *Timer* class in Figure 2 provides the functionality needed to slow down a computation.

To achieve our goals of platform independence and high performance, we chose OpenGL as our graphical foundation.

To handle OpenGL extensions conveniently, we used the OpenGL Extensions Wrangler (GLEW) library (*glew.sourceforge.net*). To interact with a *Canvas* using a mouse or keyboard, we used the GLFW library (*glfw.org*).

To ensure operability with C++11, OpenMP, and POSIX threads, we wrote TSGL in C++11, and used features of the OpenMP and POSIX thread libraries.

To create HTML-based API documentation, we used the Doxygen system (*www.doxygen.org*).

C. TSGL Implementation Issues

One issue we encountered was the frame-buffer race condition described in Section I. To address this problem, we used the *Shared Queue* parallel design pattern [3]. More precisely, each TSGL *Canvas* has its own:

- shared queue, capable of storing graphical items; and
- render-thread, responsible for rendering graphical items for that *Canvas*.

The *Canvas* class provides a variety of drawing methods, including drawPixel(), drawLine(), drawRectangle(), etc. Each has parameters appropriate for the object being drawn. Each method uses its parameters to define the graphical item being drawn, and then deposits that item in the *Canvas*'s shared queue, which is thread-safe. The render-thread retrieves the graphical items from the shared queue.

Initially, we had the render-thread drawing each graphical item to OpenGL's framebuffer. This resolved the frame buffer race condition because the render-thread was the only thread interacting with the framebuffer.

However having the render-thread do the drawing proved to be a bottleneck, so we revised the render-thread to have it convert the graphical items to textures in OpenGL's framebuffer, which the GPU then draws on the screen. The render-thread was still the only thread interacting with the framebuffer, but we were able to stress-test TSGL with 1024 threads all drawing to the same *Canvas* and maintain a full 60 frames per second display rate.

III. EXAMPLES

In this section, we present examples that show how TSGL can be used to see parallel behavior. We first examine the *Parallel Loop* pattern, and then explore the *Actor* pattern.

A. The Parallel Loop Pattern

In many programs, most of the time is spent in loop statements. This behavior is so common, programmers have created the **90-10 Rule** to describe it:

> *"90% of the time is spent in 10% of the code."*

Using parallelism to speed the processing of an otherwise slow loop is one of the parallel design patterns, known as the *Parallel Loop* pattern. In the simplest form of this pattern, the compiler generates code to (a) identify *n*, the number of available threads, (b) divide the iteration-range of the loop into *n* equal-sized chunks, and (c) give each thread one of the chunks to perform.

Explaining this behavior to students can be a challenge, even using a parallel education tool like a patternlet [1]. As we shall see, TSGL makes it possible for students to *see* the behavior of this pattern in near real-time.

1) Image Processing

For students who have grown up with smart phones and their built-in cameras, creating a "photoshop effect" to process large photographic images can be a motivating way to introduce parallelism [2]. TSGL lets us do so and see the transformation happening in near real-time.

To illustrate, Figure 3 presents a large and colorful PNG image being displayed using a TSGL *Canvas*.

Figure 3. A Colorful PNG Image

As an example, we will process this image using the *color inversion* transformation.

There are different algorithms for inverting a color image, depending on how the RGB color information is stored. If a color's RGB components are integers between 0 and 255, a pseudo-code algorithm might be given as follows:

```
Canvas canvas1, canvas2;
canvas1.loadImage(imageFile);

for each y in canvas1.getRows() {
   for each x in canvas1.getColumns() {
      Pixel p = canvas1.getPixel(x,y);
      newR = 255 - p.getR();
      newG = 255 – p.getG();
      newB = 255 - p.getB();
      canvas2.setPixel(x, y, newR, newG, newB);
   }
}
```

If the image being processed is sufficiently large and a single thread performs the algorithm, then that thread's progress may be slow enough that it can be seen in real time. For smaller images, TSGL lets us slow the processing as needed for the human eye to see the thread's progression.

In this algorithm, each pixel's value is retrieved and modified independently of all other pixel values. Since each of the loop's iterations is independent of the others, we can use the *Parallel Loop* pattern to parallelize this algorithm. The following pseudocode shows how we might do so using OpenMP's built-in mechanism for this pattern:

```
Canvas canvas1, canvas2;
canvas1.loadImage(imageFile);

#pragma omp parallel for
for each y in canvas1.getRows() {
   for each x in canvas1.getColumns() {
      Pixel p = canvas1.getPixel(x,y);
      newR = 255 - p.getR();
      newG = 255 – p.getG();
      newB = 255 - p.getB();
      canvas2.setPixel(x, y, newR, newG, newB);
   }
}
```

Suppose that there are 800 rows on our *Canvas*, and that we are running this on a quad-core CPU. Then when execution reaches the OpenMP **#pragma** directive, OpenMP will divide the 800 iterations of the outer for loop into four chunks (0-199, 200-399, 400-599, and 600-799), and give each chunk to a different thread.

Figure 4 is a screenshot of four threads inverting the image from Figure 3, with the computation about 2/3 done:

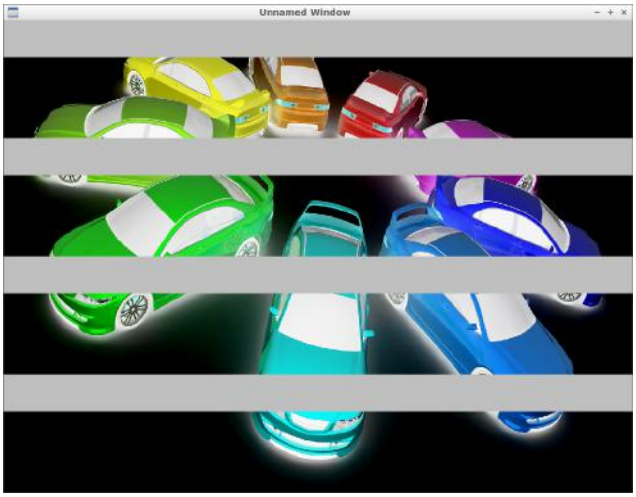

Figure 4. Color Inversion Using Four Threads; In Progress

The four gray bands in Figure 4 are the portions of the image that were unprocessed at the time the screenshot was taken; the other areas' pixels had been inverted. Note that the four gray bands are all equal in size, indicating that each thread has an equal amount of work remaining. Put differently, each thread has made an equal amount of progress on its chunk of the image. TSGL thus lets a student see: (i) *what* work each thread is doing; (ii) *when* that work is being done, in relation to the other threads' work; and (iii) *how fast* each thread is working, compared to its peers.

At the end of the loop, we have each thread use a unique color to draw a rectangle around its chunk of the image, so that its contribution toward the overall computation can be clearly seen, as shown in Figure 5.

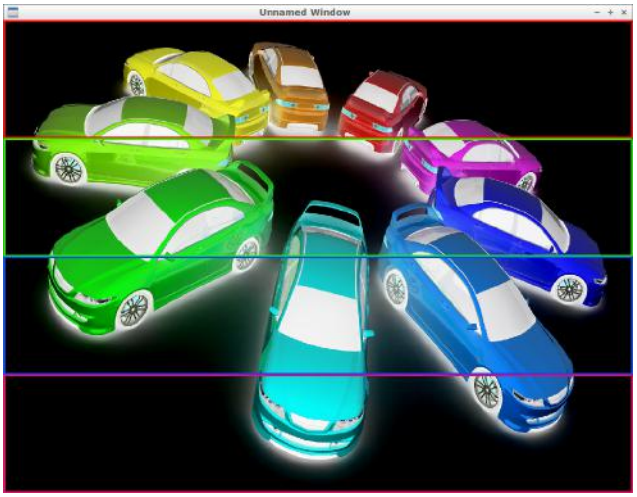

Figure 5. Color Inversion Using Four Threads; Finished

TSGL makes it possible to provide students with a sequential, graphical version of any of the common image transformations (e.g., color-to-grayscale, sepia tinting, resizing, brightening, sharpening, blurring, etc.) and have the students time the operation. If they then parallelize the operation's processing loop and rerun the program, they will see and experience the difference in speed and behavior between the original sequential version and their parallel version. By manually varying the number of threads in such a parallel program and seeing the result, first-year students can develop an intuitive understanding of abstract concepts like the *Parallel Loop* pattern, scalability, and so on.

2) Numerical Integration

For CS 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 algorithm to compute the integral of **`f(x)`** from **`a`** to **`b`** using the "rectangle method" might be given as follows:

```
heights = 0.0;
recWidth = (b-a) / NUM_RECTANGLES;
halfRecWidth = recWidth / 2.0;
for (i = 0; i < NUM_RECTANGLES; i++) {
   xLo = a + i * recWidth;
   xMid = xLo + halfRecWidth;
   y = f(xMid);
   heights += y;
}
return heights * recWidth;
```

For each rectangle, the algorithm's loop accumulates the sum of the rectangles' "heights" (from the rectangle's midpoint on the x-axis up to the curve) in the variable **`heights`**. When the loop is completed, we multiply those accumulated "heights" by the width of the rectangle to compute the return value.

To convert this algorithm into a parallel algorithm, we can again use OpenMP and the *Parallel Loop* pattern. To help students see how the algorithm works, we can use TSGL to (i) give each thread a color, and (ii) have the thread draw its rectangles, as shown in the following pseudo-code:

```
heights = 0.0;
recWidth = (b-a) / NUM_RECTANGLES;
halfRecWidth = recWidth / 2.0;
CartesianCanvas canvas(MAX_X, MAX_Y);
canvas.showAxes();
canvas.drawFunction(f);
#pragma omp parallel reduction(+:heights)
{
 threadID = getThreadID();
 Color color = canvas.getMyColor(threadID);
 #pragma omp for
 for (i = 0; i < NUM_RECTANGLES; i++) {
   xLo = a + i * recWidth;
   xMid = xLo + halfRecWidth;
   y = f(xMid);
   canvas.drawRec(xLo, 0, xLo+recWidth, y, color);
   heights += y;
 }
}
return heights * recWidth;
```

That is, we (1) create a *CartesianCanvas* object that all threads will share; (2) tell that canvas to display its axes; and (3) tell that canvas to draw the function we are integrating. We then (4) direct OpenMP to create a parallel block, launching new threads; (5) have each thread retrieve its id number; (6) use that id number to give each thread a unique color; and (7) direct OpenMP to divide the iterations of the for loop among the threads launched in step 4. Within the loop, (8) each thread draws the current rectangle on the canvas, using its unique color.

Figure 6 shows two screenshots of a running TSGL implementation of this algorithm, using a quarter of the unit circle function for **`f()`** and one thread. In the left shot, **`NUM_RECTANGLES`** is 10; its value is 100 in the right shot.

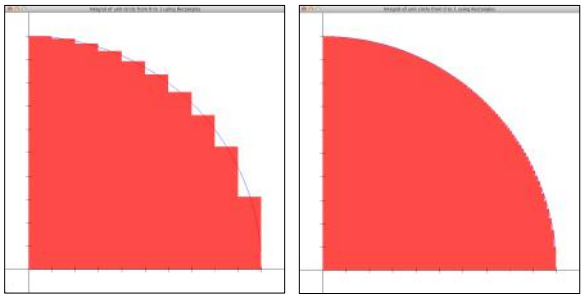

Figure 6. Integration With One Thread:
(a) 10 Rectangles; (b) 100 Rectangles

Since a single thread is performing the integration, each rectangle is drawn using the same color (red). This program lets us specify the number of rectangles and threads from the command-line, so Figure 7 shows the same computation using 10 rectangles with two vs. four threads:

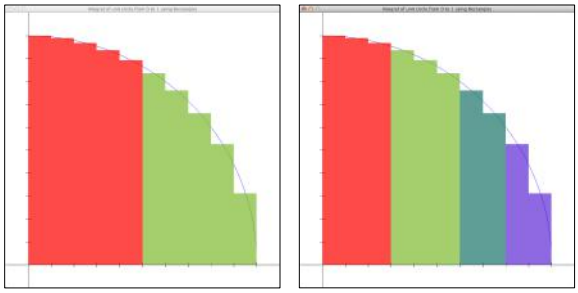

Figure 7. Integration With 10 Rectangles:
(a) Using Two Threads; (b) Using Four Threads

As before, TSGL lets us see how the *Parallel Loop* pattern works. Figure 7a shows that for two threads, the pattern divides the iteration range into two contiguous "chunks"; Figure 7b shows that for four threads, it divides the range into four such "chunks". Since each thread is coloring its "chunk" using its unique color, we can infer (and verify) that for *n* threads, the pattern divides the iteration range into *n* contiguous "chunks". It is also easy to see how this pattern divides the iterations when they are not evenly divisible by the number of threads, as shown in Figure 7b.

By letting us color-code each thread differently, TSGL lets us readily see how a computation's workload is being divided among its threads.

3) The Mandelbrot Set

The Mandelbrot set is a well-known fractal figure. A pseudo-code algorithm to draw it might be simplistically given as follows:

```
Canvas canvas(MAX_X, MAX_Y);
for (y = 0; y < MAX_Y; y++) {
   for (x = 0; x < MAX_X; x++) {
      Color color = mandelColor(x, y);
      canvas.drawPoint(x, y, color);
   }
}
```

In this algorithm, we have hidden the computation of whether a given (x,y) point is in the Mandelbrot set within a **`mandelColor()`** function. This function is sufficiently time-consuming (i.e., it contains another loop) that on many computers, one can see the individual rows of the figure being drawn. On newer computers, TSGL lets us slow the computation sufficiently for this to occur.

As with the integration example in Section 3.1, the *Parallel Loop* pattern can be used to divide the iterations of this algorithm's outer loop into "chunks" performed by different threads.

However, unlike our previous examples, the time to process an (x,y) point in the Mandelbrot figure can vary widely, depending on the (x,y) point being computed. TSGL lets students see this time-variance; if we use the *Parallel Loop* pattern and 8 threads, we observe behavior like that shown in Figures 8 and 9:

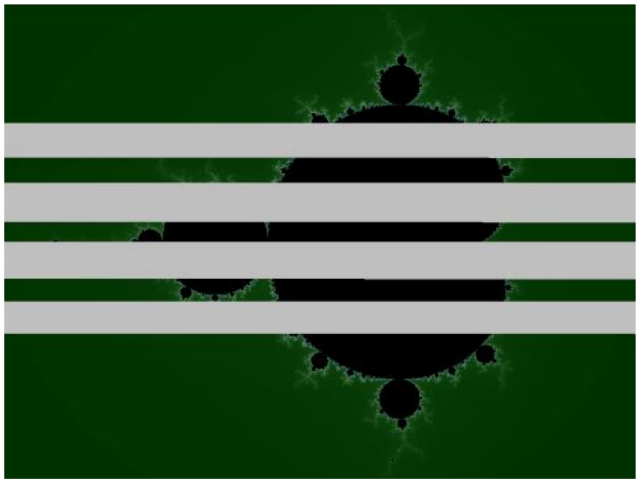

Figure 8. Mandelbrot Using Eight Threads; In Progress

Like Figure 4, Figure 8 shows the figure in an intermediate state. However, where the threads in Figure 4 had made equal progress, the threads in Figure 8 have not. More precisely, threads 0 and 1 have completed their chunks (the top two eighths of the figure), and threads 6 and 7 have completed their chunks (the bottom two eighths), but threads 2, 3, 4, and 5 are still working on their respective eighths, as indicated by the four gray bands in the middle of the figure.

This behavior occurs because points within the Mandelbrot set (the black portion of the figure) take longer to compute than points outside the set. Since the threads drawing the middle rows have more inside-the-set points to compute, it take them longer to complete their "chunks". TSGL's near real-time drawing can thus let students see the effects of non-uniform workloads, which can be used to motivate the introduction of other parallel design patterns that do a better job of load balancing across the threads.

When finished, we have each thread draw a colored rectangle around the portion of the figure it drew, as shown in Figure 9.

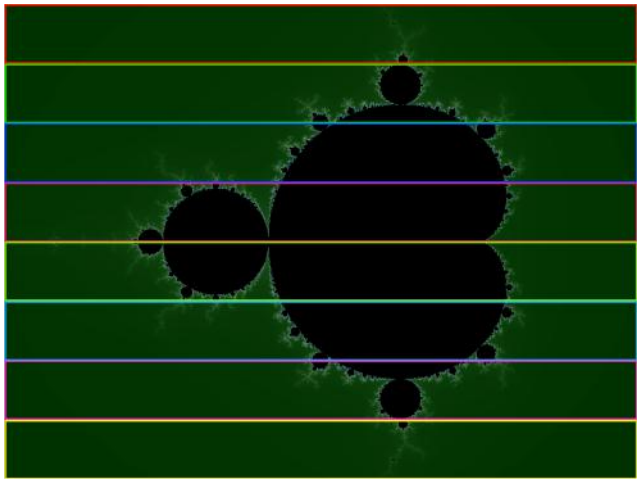

Figure 9. Mandelbrot Using Eight Threads; Finished

B. *The Actor Pattern*

Another parallel design pattern is the *Actor* pattern, in which autonomous actors perform their prescribed behaviors. In the remainder of this section, we present an example of this pattern: the classic Dining Philosophers problem.

1) *The Dining Philosophers*

In the Dining Philosophers problem, *n* silent philosophers sit around a table, with a large bowl of food in the middle. There are *n* chopsticks on the table, each positioned between a pair of philosophers. A philosopher may be thinking, hungry, or eating, and may think for an arbitrary length of time before becoming hungry, but in order to eat, he or she must acquire both of the adjacent chopsticks. As shared resources, the chopsticks represent a potential source of race conditions. The problem is to devise a strategy that all philosophers can follow, that ensures (a) no deadlock occurs, (b) no livelock occurs, (c) that no philosopher starves, and (d) that a philosopher who is thinking does not prevent a philosopher who is hungry from eating.

Using the Actor pattern, each philosopher has its own thread that performs the strategy for that philosopher. TSGL makes it possible to create a visualization to help students see a strategy running in near real-time, using color-coding to represent a philosopher's state. Figure 10 provides a proof-of-concept visualization, in which the large central gray circle represents the table, the five medium-sized circles represent the philosophers, and the five small circles represent the chopsticks:

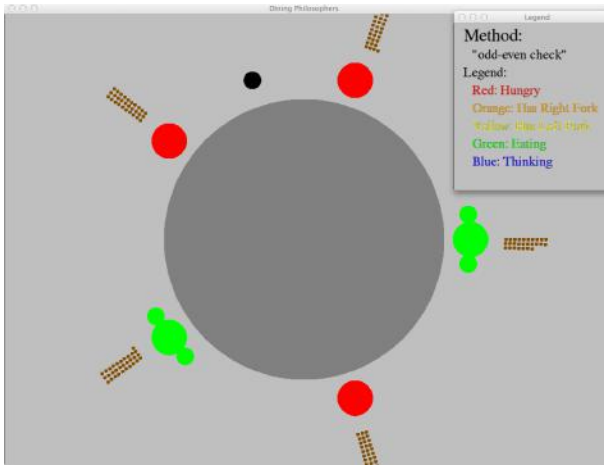

Figure 10. Visualizing the Dining Philosophers (1)

The separate "Legend" window notes the strategy being used and the state each color represents.

At the moment shown in Figure 10, the bottommost and the two topmost philosophers are all hungry (as indicated by their red color) but neither has picked up the (black) chopstick between them, because the other two (green) philosophers are holding the other chopsticks they need. The small brown dots "behind" each philosopher indicate the number of times that philosopher has eaten, allowing us to see at a glance that no philosopher is starving.

Figure 11 presents a screen capture of the same program later on, showing one of its intermediate states:

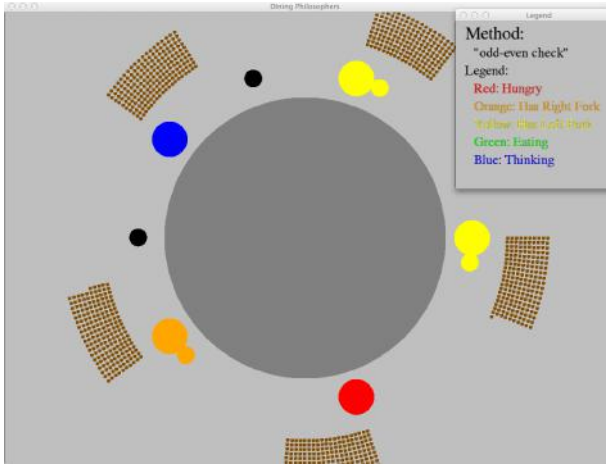

Figure 11. Visualizing the Dining Philosophers (2)

In Figure 11, no philosopher is eating at the moment. Having acquired its left chopstick, the topmost (yellow) philosopher is about to eat, as soon as it picks up its right chopstick, which is available. Its clockwise (yellow) neighbor also has its left chopstick, but is unable to eat until the first philosopher releases its right chopstick. Its clockwise (red) neighbor is hungry, but is unable to eat as neither of its chopsticks are available. Its clockwise (orange) neighbor is about to eat, having acquired its right chopstick, and is about to pick up its left chopstick. Again, the brown dots "behind" each philosopher indicate that no philosopher is starving, and that each is eating as often as its peers.

TSGL can also be used to help students see incorrect strategies. For example, Figure 12 shows a version of the program in which the philosophers follow a strategy that leads to deadlock:

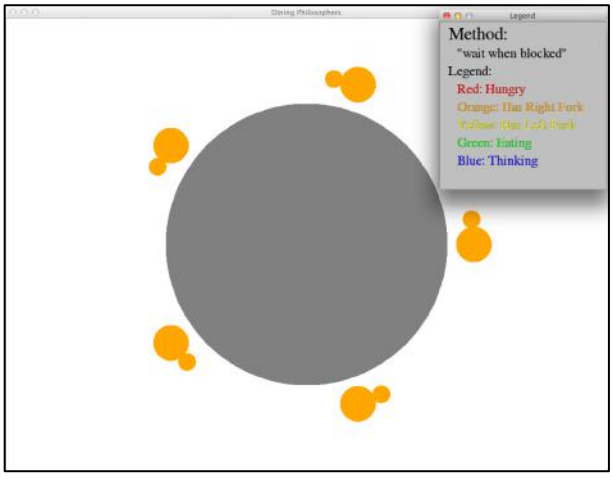

Figure 12. Visualizing the Dining Philosophers: Deadlock

In Figure 12, each philosopher has acquired its right fork and is waiting to acquire its left fork, producing a circular wait.

TSGL thus makes it possible to create visualizations for "classic" synchronization problems like the Dining Philosophers, Producer-Consumer, Readers-Writers, Sleepy Barber, and so on.

IV. Discussion

We have seen that TSGL makes it possible to visualize parallel computing patterns like the *Parallel Loop* and the *Actor* patterns. This raises the question: Can TSGL be used to create visualizations that help students understand other parallel concepts?

We believe the answer to this question is "Yes" and that the potential of TSGL is mainly limited by our creativity. For example, we have begun work on a visualization of the *Task Queue* pattern, as follows:

a. Open a *Canvas* whose background is white and whose width is proportional with *m*, the length of the queue;
b. On the *Canvas*, draw *m* black rectangles, each representing one of the queue's tasks, saving a reference to each rectangle in a shared queue.
c. Each time a thread gets a task from the task queue, use the shared queue to change the color of the corresponding rectangle from black to white.

At the outset, the *Canvas* will show *m* black rectangles, but as tasks are removed from the task queue, the corresponding rectangles will 'disappear' into the white background. The result will be a kind of "reverse progress bar" that grows shorter as the length of the task queue decreases.

Alternatively in step (c), a thread could change the color of the rectangle to that thread's unique color. At the end, the distribution of colored rectangles on the *Canvas* would reflect the distribution of tasks among the threads.

As a second example, we have begun work on a TSGL visualization of the parallel Merge Sort algorithm. This and similar visualizations will let students see how the sequential and parallel versions of an algorithm differ.

We believe that a key to creating effective visualizations is *scalability* – making certain that a student can change the number of threads being used without recompiling, and that the visualization's behavior changes accordingly, in both appearance (see Figure 7) and execution-time. Each of the examples described in Section III feature scalability, as this allows a student to explore the program's behavior using differing numbers of threads. TSGL's near real-time graphics let a student actually *see* a scalable program run faster as the number of threads is changed from 1 to 2 to 3 to 4 to … We have used it in both a lecture setting and a lab setting; seeing a program run faster motivates students to quantify the speedup and determine precisely how much faster the program is running. TSGL appears to have great potential as a parallel pedagogical tool.

V. Conclusions

In keeping with the adage *"A picture is worth 1000 words,"* we believe that an effective way to teach students about parallelism is to show them interactive visualizations that illustrate abstract parallel concepts. Such visualizations will provide students with memorable mental imagery, and the interactivity will let them explore the parallel behavior and master the concept.

To help ourselves and others create such visualizations, we have created the *thread-safe graphics library* (TSGL). TSGL is an object-oriented library whereby C++11, POSIX, and/or OpenMP threads can safely draw on the same *Canvas* object, and one can see the results in near real-time.

To illustrate the use of TSGL, we have presented several examples, including image processing, numerical integration, the Mandelbrot Set, and the Dining Philosophers Problem. We have also described other visualizations on which we are currently working; we believe TSGL has great potential as a teaching tool.

For those who would like to try it, the current release of TSGL may be freely downloaded from GitHub (see https://github.com/Calvin-CS/TSGL) under the GNU Public License (v. 3). It was developed on Ubuntu Linux 14.0.4 and has been successfully tested on both MacOS X Yosemite and Windows 7. The project's Github site includes installation instructions, tutorials on how to use the library, and its API. We look forward to seeing new and exciting visualizations that others will create using TSGL.

Acknowledgment

We wish to thank the National Science Foundation, whose grant NSF-DUE #1225739 made TSGL possible.

References

[1] J. Adams, "Patternlets: A Teaching Tool for Introducing Students to Parallelism," 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW-15), Hyderabad, India, May 2015. p. 752-759. DOI=10.1109/IPDPSW.2015.18.

[2] S. Massung and C. Heeren, "Visualizing Parallelism in CS2", Third NSF/TCPP Workshop on Parallel and Distributed Computing Education (EduPar-13), May 2013. Accessed 2015-01-10. Online: http://grid.cs.gsu.edu/~tcpp/curriculum/sites/default/files/Visualizing%20Parallelism%20in%20CS%202_0.pdf

[3] T. Mattson, B. Sanders, B. Massingill. *Patterns for Parallel Programming*, Pearson Education, 2005.

[4] S. Prasad, et al., "NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing – Core Topics for Undergraduates," Dec 2012. Accessed 2015-01-10. Online: http://grid.cs.gsu.edu/~tcpp/curriculum/sites/default/files/NSF-TCPP-curriculum-version1.pdf

[5] M. Sahami, et al., "Computer Science Curricula 2013," Online: http://www.acm.org/education/CS2013-final-report.pdf. Accessed 2016-01-10. DOI=10.1145/2534860.