Easel: A Java Based Top-Down Approach to 3D Graphics Education

Philip J. Rhodes and Baoqiang Yan

Department of Computer and Information Science, University of Mississippi, Oxford, MS USA

Abstract

We describe Easel*, a simple 3D graphics pipeline implementation targeted toward undergraduate Computer Graphics education. Easel is an interactive system written entirely in Java, which presents unique challenges and opportunities for teaching not only 3D Graphics, but also a well-reasoned approach to software engineering and development. Achieving a reasonable frame rate in addition to correct results requires students to think carefully about performance, ease of implementation and maintainability.*

Categories and Subject Descriptors (according to ACM CCS): Computers and Education [K.3.2]: Computer Science Education—

1. Introduction

With the advent of fast graphics cards, many instructors use OpenGL, Direct3D, or Java3D to teach introductory 3D Computer Graphics. This choice is reasonable, since most graphics programming is now done using these popular graphics APIs. However, using an API allows a student to get impressive results without fully understanding the underlying algorithms, costs, and implementation.

Another approach is to have students implement traditionally static rendering methods such as *ray tracing* or perhaps *radiosity*. In order to correctly implement the algorithm, the student must have a deep understanding of how the algorithm works. However, the static nature of such algorithms may fail to engage students who are very used to the animated, highly interactive graphics environments providing by video games.

A third approach, described in this paper, is to have students implement a graphics pipeline themselves, without the aid of OpenGL or similar APIs. In addition to engaging the students with an interactive environment, the *frame rate* metric, familiar to any gamer, allows students to easily see the impact of design and implementation decisions.

It is presumed that many educators have taken a similar approach in their own courses, but we believe that using Java as the implementation language is unusual. Java presents some advantages over C/C++, in that Java programs are usually easier to debug, and are compatible across platforms. Java has also become extremely popular as the main language of instruction in many departments.

The *Easel* system is written entirely in Java, performing all graphics operations, including rasterization, without the help of other packages. Although the relative speed of Java and C++ is hotly debated, it is certain that writing efficient Java code requires awareness of Java-specific issues like garbage collection, in addition to minimizing computation. In order to maximize the frame rate, interesting decisions must be made with regard to performance, maintainability, and ease of implementation. In short, implementing a graphics system like Easel is an excellent application of an undergraduate's understanding of Computer Science theory and practice.

Over several iterations of an introductory Computer Graphics course, students have been asked to implement most of the Easel pipeline, from constructing basic shapes all the way through to Gouraud shading. The course is very implementation intensive, and fits well in a curriculum seeking to provide students with practical experience working with a larger codebase.

This paper shares knowledge gained from using Java to implement a simple graphics pipeline in an educational setting. We describe the technical challenges that must be ad-

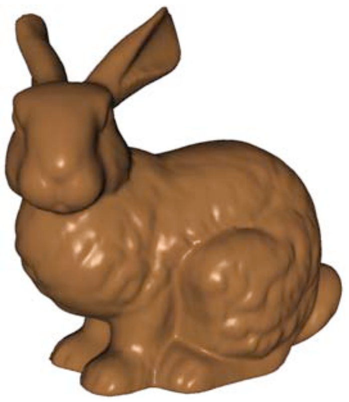

Figure 1: *The Stanford Bunny rendered with Gouraud Shading using the Easel system. A current desktop machine can render this model at a rate of least 10 frames per second.*

dressed in order to achieve reasonable performance in Java while still producing clear and maintainable code. Using the techniques described in this paper, it is possible to render models like the Stanford Bunny, consisting of about 69,451 triangles, at about 10 frames per second on current hardware.

2. Background

The relative merits of the *bottom-up* and *top-down* approach to Computer Graphics instruction has been a subject of some debate in recent years. The bottom-up approach is more traditional, and emphasizes fundamental Graphics algorithms such as shading, clipping, and rasterization. The top-down approach is a more recent response to the appearance of 3D Graphics APIs like OpenGL and Direct3D.

A discussion panel at SIGCSE '06 presented several points of view on this topic [ACSS06]. In that discussion, Edward Angel favored a top-down approach using OpenGL. He argues that it is not strictly necessary to know the lowest level details in order to use OpenGL effectively. His textbook [Ang08] follows this philosophy.

Sung and Shirley argue that the top-down approach is well suited to a single graphics course with mature students [SS03]. They point out that students that are already in industry probably don't need experience with large projects, and instead require the skills that will be most immediately useful in their career. They describe a ten week course that teaches 2D interactive graphics programming. Rather than restrict themselves to a single API, Sung and Shirley argue for examining several different APIs throughout the course. They warn against the risk that students will feel that "if the API doesn't support it, it can't be done".

However, during the SIGCSE discussion [ACSS06], Peter Shirley claims that students achieve a deeper understanding of Graphics fundamentals when they must implement rasterization in a bottom-up approach. If a two course sequence is possible, Shirley argues that the first course should teach low-level fundamentals, while the second course teaches more advanced material using OpenGL as the underlying graphics system. We agree with this assessment, and the course described here is meant to be the first of a two course sequence.

Steven Cunningham also argues for a two course sequence [Cun98a]. However, he would start with the high level material, perhaps using [Ang08] as a text, and explore the fundamental algorithms in the second course. Cunningham has also emphasized the need to teach students Human-Computer Interaction and related topics in CS curricula, perhaps at the expense of pixel-level algorithms [Cun98b]. However, Cunningham points out that Computer Graphics can be a great application area to demonstrate the importance of efficiency. In fact, several authors specifically mention that Computer Graphics works very well as a motivating application of basic Computer Science concepts. We agree strongly, and feel that the bottom-up approach is very good at reinforcing the knowledge that students have learned in *Analysis*, *Compilers*, and *Computer Architecture* courses. Of particular interest to us is the various design issues that arise, forcing students to think carefully about how to balance performance with factors like clarity, robustness and maintainability.

3. Programming in Java

Java has become a very popular language in Computer Science curricula, mainly due to the fact that it is easier to debug Java programs compared to C and C++. Common errors such as out of bounds array accesses are immediately caught by the Java Virtual Machine (JVM), generating an exception message that tells the programmer which line of source code caused the problem, and the methods on the stack when the problem occurred. In contrast, a similar bug in a C++ program will often go undetected. In the worst case, writing past the end of an array allocated on the stack will overwrite other local variables, perhaps causing a truly mystifying and frustrating bug elsewhere in the program. Many educators would argue that learning to debug even these pathological cases is important for students. Others argue that the frustration resulting from such behavior is an impediment to the learning process, especially in courses like Graphics, where the focus is on other material.

Java's garbage collection is perhaps its most famous distinguishing feature, freeing programmers from having to manually deallocate space for objects that are no longer used. However, the garbage collection operation can be time consuming, and programmers must pay careful attention to this cost when implementing performance-sensitive software in Java.

The comparative speed of Java versus compiled languages

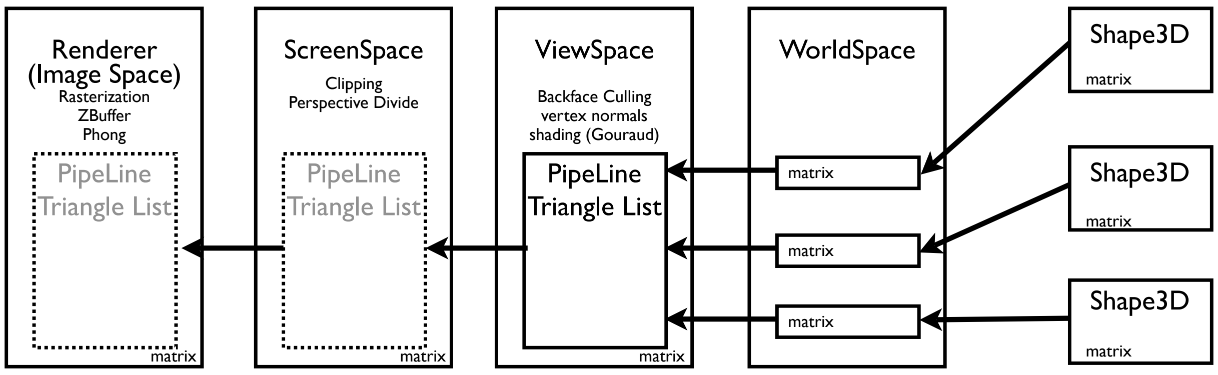

Figure 2: *The Easel Pipeline.*

such as C and C++ has been a topic of intense debate for years. Programming language partisans appear able to produce test results in support of their favorite language. The purpose of this paper is not to weigh in on this debate, but rather to describe the methods used to implement a Java based graphics pipeline with reasonable performance.

3.1. Java programming for performance

Garbage collection in Java has been a topic of considerable research, and recent versions of the JVM provide several options for how garbage collection should be performed. The JVM will make an effort to choose an appropriate garbage collector for the machine it is running on. For example, a JVM running on a machine with several cores or processors will likely use concurrent garbage collection, where a separate thread periodically cleans up unused memory [Sun]. There are also variations that perform *generational* garbage collection, paying special attention to memory that was recently allocated, since this storage is more likely to be garbage than memory that has been in use for a long time. In any case, garbage collectors are sensitive to the number of objects a program allocates on the heap, since each object must be properly classified as garbage or non-garbage.

Regardless of the type of collector used, it is always advantageous to reduce the frequency with which collection should be performed, and to make the collection process as quick as possible by minimizing the amount of memory that must be reclaimed. With this in mind, students are given the following two suggestions:

S1. *Reuse objects whenever possible.*

S2. *Try to allocate a few large objects instead of many small objects.*

The first suggestion is especially applicable when a temporary object is needed for a calculation that is performed repeatedly. For example, when transforming a vertex using a matrix, it is convenient to use an extra array of four elements as scratch space. Rather than allocating this array object at the beginning of each call to the matrix.transform() method, we suggest using (and re-using) a data member of the matrix class. Although accessing data members can be more expensive than accessing local variables, this penalty is tiny compared to the expense of repeatedly allocating and garbage collecting scratch space.

The second suggestion is meant to reduce the time spent by the garbage collector when examining objects on the heap. Even if no new garbage is being generated, a large number of heap objects will increase garbage collection time because each object must still be examined by the garbage collector. Depending on the variety of garbage collector used, this can cause noticeable pauses in the animation which not only lower the frame rate but also annoy the user.

Two more suggestions address computational load. One source of load is associated with method calls. All methods are effectively *virtual* in Java, meaning that in many instances the code to be executed cannot be determined until runtime. Associated chores such as writing the return address to the stack and passing arguments adds additional overhead. In some cases, these costs can be avoided by labelling a method with Java's `final` keyword. This keyword tells the compiler that no child class will override the method, so it is known at compile time what code will be executed. Methods that are declared final are candidates for *inlining*, meaning that a call to the method will be automatically replaced with equivalent bytecode, avoiding the costs of a method call entirely. Not all code is suitable for inlining, but this technique works especially well for short accessor methods. Our third suggestion is:

S3. *Avoid an excessive number of method calls, and use the* `final` *keyword when appropriate.*

Another source of computational load is argument checking. It is widely considered good practice to perform a safety check on method arguments, especially when invalid arguments would make the state of an object invalid as well. However, there are many instances where an invalid argument does not make an object inconsistent, and where any safety check would be either redundant or needlessly repeated. An example of redundancy is a check for array indices that might be out of bounds. In C and C++, such a check is an essential part of writing secure code. However, the Java runtime environment already checks array indices and generates an exception when they are out of bounds, so an additional explicit comparison is wasteful. A more efficient approach is to catch the exception generated by the JVM and perform the appropriate action.

For an example of a safety check that is needlessly repeated, consider a method `getTriangleCoords(int ti, float [] coords)` that fills a nine element array with the coordinates of the triangle specified by first argument. During rasterization, this method is called for every triangle in the scene, and is passed the same array every time. Verifying that the array has a length of at least nine imposes a computational burden that is not useful. In the spirit of these observations, we offer a fourth suggestion:

S4. *Avoid excessive checks for argument correctness.*

These last two suggestions can be seen as "dangerous", in that they could be misunderstood by the student. Suggestion three could be taken as license to write very long methods, spanning several screenfuls of text. In fact, the suggestion is meant to encourage the writing of methods that work on large collections of data, so that the number of method invocations is reduced. For example, rather than writing a method that transforms a single vertex with a matrix, we prefer to wrap that code in a loop so that it operates on an entire array of vertices. If suggestion two has also been followed, storing data together in large objects, this programming style will come naturally.

The fourth suggestion could also be misconstrued to mean that argument checking need never be done. Rather, it should be done only once, at the appropriate place in the code. The java *package* mechanism provides a natural place for such checking. A Java package is a collection of classes that together provide a particular functionality. Class methods that are not explicitly labeled `private`, `protected`, or `public` are assumed to have *package* access, meaning that only classes in the same package can call them. In the interest of performance, we can perform safety checks only for public methods that can be called directly by the user. Since package access methods are only called from inside the Easel package, we assume they are being passed reasonable argument values.

Implicit in this assumption is the hope that the Easel package contains no bugs, which is unlikely to be true. Clearly, we are trading some reliability for performance. Students are encouraged to reason carefully about this tradeoff throughout the course, practicing real engineering by deciding on the right balance between performance, reliability and maintainability.

```
while(true){
    s1.setYRotation(theta);
    renderer.render();
    theta += 0.01;
}
```

Figure 3: *A very simple main loop for an Easel program. A shape s1 is made to spin around the Y axis in object space. The loop will continue until the application is quit by the user.*

4. The Easel Design

The Easel system is designed as a straightforward graphics pipeline, as shown in figure 2. The various stages of the pipeline are explicitly represented as objects containing methods that operate on collections of triangles maintained in the *TriangleList* and associated child classes. For each frame, the *Renderer* class requests a TriangleList containing all the triangles of the scene. This request ripples down the pipeline, eventually causing a single *PipeLineTriangleList* to be assembled from the various shapes in the scene. The PipeLineTriangleList is then passed back through the pipeline, transformed by matrices, and operated upon at each stage. When it returns to the Renderer stage, the triangles are rasterized to the imagebuffer.

4.1. The Easel Renderer

The process described above is driven by successive calls to the `render()` method of the Renderer object that terminates the pipeline, where each call to this method results in the drawing of a single frame. The `render()` method is typically called inside a loop in `main()`, which terminates only when the main thread is killed by closing the rendering window. Inside this loop, the programmer is free to modify object position, rotation and scaling in either object or world space, and to modify camera parameters as desired. Figure 3 shows a simple example.

If the renderer simply drew directly to the screen, the quality of the resulting animation would be poor, due to flickering caused by simultaneous reading and writing to the display memory. *Double buffering* solves this problem by using two separate blocks of display memory. Although Java's *Swing* environment supports double buffering, we had trouble getting satisfactory results with the supplied functionality, though we haven't tried with the most recent JVM version. However, in the later weeks of the course, we require direct access to the memory representing our framebuffer, allowing us to perform our own rasterization. For this reason, we opted to implement our own double buffering using Java's *Image* class. That is, we draw to a Java Image object,

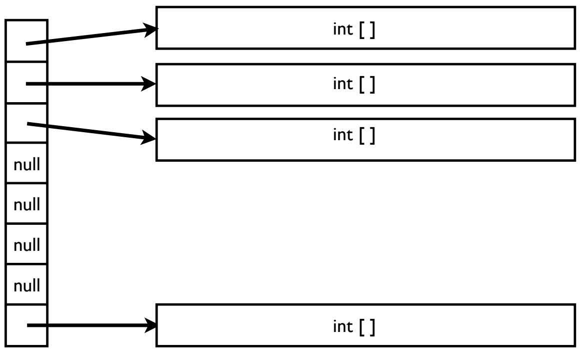

Figure 4: *The IntList Implementation. Data is stored in several arrays called* banks*, each of the same length. Banks can be created as needed.*

and when the scene is completely rendered, we draw the Image to the window. Image objects can be constructed using a previously allocated array of bytes, allowing us the direct access to memory we require.

This scheme is implemented using two threads. The *render thread* runs the renderer and pipeline code, while the *drawing thread* does the drawing to application window. Although not best practice, we find that the user interface thread that is already present in Swing applications can be used without trouble as the drawing thread. The main application thread is used as the render thread. Activity between the two threads is coordinated using Java's built in locking mechanism, preventing simultaneous access to the Image. Using a boolean, we also ensure that each frame is drawn only once. The result is a smooth and flicker free animation of the scene.

4.2. TriangleList

The representation of triangles and vertices is probably the design decision that affects performance most, since most stages of the pipeline must repeatedly access triangles, vertices, and associated data like normals and vertex colors.

Perhaps eager to apply the software engineering knowledge learned in other classes, students are sometimes tempted to wrap each individual vertex and triangle in its own object. Because Java implements arrays of objects as arrays of object references, this strategy results in very poor performance, for the reasons outlined in section 3.1. Instead, we represent the vertices and triangles as large one-dimensional arrays of floats and ints, respectively.

In the initial offering of the course, TriangleList was implemented as a set of fixed length arrays. Package classes were able to directly access these arrays, which produced good performance but also code which was difficult to debug and maintain. Figure 5 shows an example. Worse, the fixed length of the arrays presented problems during the clipping stage, which increases the number of triangles. For some unusual scenes, the reduction in triangle count produced by backface culling does not provide enough room for the extra triangles produced by clipping. Although this can be addressed by increasing the initial size of the arrays by some percentage, the solution is not elegant.

A later offering of the course introduced new Java classes to solve the problems associated with fixed length arrays. The *ByteList*, *FloatList*, and *IntList* classes can represent lists of basic types of practically unlimited length. These classes exploit the fact that Java implements two dimensional arrays as an array of pointers to one dimensional arrays. Figure 4 illustrates that we can dynamically allocate the one dimensional arrays (called *banks*) as needed, making it unnecessary to know in advance the maximum space needed. Assuming a bank size of 1024 bytes, and 1024 pointers, an IntList can store up to 1GB of data, which is more than sufficient for our purposes. Lastly, we do not de-allocate banks between frames, but overwrite existing data instead, so allocation costs are paid mainly during the rendering of the first frame.

The first implementation using the List classes described above provided methods for setting and getting single bank elements. However, performance was unacceptable, due to the large number of method calls, and repeated calculations associated with choosing the correct bank for retrieval. This situation was partially addressed by augmenting the List classes to allow access to three elements at one time, greatly reducing the number of method calls, and simplifying the code. Comparing figure 6 to figure 5 demonstrates the simplicity and clarity of the new implementation. Unfortunately, this clarity comes at the expense of performance. The frame rate drops by approximately 30% for larger models like the Stanford Bunny, though the drop is a more modest 2% for a Utah Teapot model with only 2256 triangles. This is a good example of the engineering tradeoffs discussed in the course. Should we choose the clearer, more robust implementation that produces safer, easier code, or deliver increased performance to the user at the expense of development and debugging time? Because many students are very familiar with the frame rate metric through gaming experience, the question has some relevance to them. Many want to write code that is as fast as possible, putting themselves in the position of the gaming user. When more safety-critical scenarios are considered, such as a medical application, students agree that a cleaner design is essential.

5. Performance and Profiling

Ideally, we'd prefer not to have to make the sort of tradeoffs described in the previous section, and would like to code that is both clean and fast. *Profiling* is an important tool in the pursuit of performance, and is available via a command line argument to the JVM. After execution has completed, a report is printed to the console detailing the amount of time

```
/** Copies the coordinates of the three vertices of the specified triangle
*  into the coords array.  The array must have length of at least nine elements.
*  A reference to the array is returned as a convenience.
*/
public float [] getTriangleCoords(int ti, float [] coords){

    coords[0] = this.vertices[ 3 * this.triangles[ti * 3 ]     ];
    coords[1] = this.vertices[ 3 * this.triangles[ti * 3 ] + 1 ];
    coords[2] = this.vertices[ 3 * this.triangles[ti * 3 ] + 2 ];

    coords[3] = this.vertices[ 3 * this.triangles[ti * 3 + 1]     ];
    coords[4] = this.vertices[ 3 * this.triangles[ti * 3 + 1] + 1 ];
    coords[5] = this.vertices[ 3 * this.triangles[ti * 3 + 1] + 2 ];

    coords[6] = this.vertices[ 3 * this.triangles[ti * 3 + 2]     ];
    coords[7] = this.vertices[ 3 * this.triangles[ti * 3 + 2] + 1 ];
    coords[8] = this.vertices[ 3 * this.triangles[ti * 3 + 2] + 2 ];

    return coords;
}
```

Figure 5: *The plain array implementation of a method to retrieve the coordinates of a triangle. Though this implementation is fastest, students found it confusing and difficult to debug.*

```
/** Copies the coordinates of the three vertices of the specified triangle
*  into the coords array.  The array must have length of at least nine elements.
*  A reference to the array is returned as a convenience.
*/
public float [] getTriangleCoords(int ti, float [] coords){

    this.triangles.get(3 * ti, this.triangle);

    this.vertices.get3( 3 * this.triangle[0], coords, 0);
    this.vertices.get3( 3 * this.triangle[1], coords, 3);
    this.vertices.get3( 3 * this.triangle[2], coords, 6);

    return coords;
}
```

Figure 6: *The IntList/FloatList implementation of the same method shown in figure 5. The code is much simpler, but at some cost to performance.*

```
     Compiled + native   Method
  7.7%   406  +     0    edu.olemiss.cs.graphics.PerspectiveFlatShadeViewSpace.getTriangleList
  3.4%   182  +     0    edu.olemiss.cs.graphics.ScreenSpaceMatrix.transform_to_homogeneous
  2.5%   135  +     0    edu.olemiss.cs.graphics.Matrix.transform
  2.5%   135  +     0    edu.olemiss.cs.graphics.PipeLineTriangleList.doPerspectiveDivide
  :
  :  <ellided>
  :
   22.4%  1186  +     0    Total compiled

         Stub + native   Method
 67.0%     0  +  3554    apple.awt.CRenderer.doPoly
  7.9%     0  +   421    java.lang.StrictMath.floor
 74.9%     0  +  3975    Total stub
```

Figure 7: *This partial listing of the profile for the Renderer using Java's 2D graphics shows that the doPoly() method, an implementation of the Java drawPolygon() method, is taking up 67% of the execution time on the render thread.*

Table 1: *Frame rates for various models on a 2.6 GHz 2 x Dual Core Intel Xeon Macintosh. Gouraud Rendering and z-buffer were implemented using a hand written rasterizer.*

Model Name	Number of Triangles	Frame Rate
Teapot	2256	42.7
XWing	6124	36.5
Bunny	69451	11.3
Dragon	871414	1.4

spent in each method by each thread. Profiling is used during lecture to explore design decisions, and students are encouraged to use this important tool during development.

Early in the course, we use a Renderer implementation that uses Java's 2D graphics. In order to implement features like z-buffer and Gouraud and Phong shading, we are forced to move to our own *Digital Differential Analyzer (DDA)* rasterization implementation. In addition to these features, we can also motivate the need for our own rasterizer using profiler data. The profile in figure 7 shows that a renderer using Java's `drawPolygon()` method spends 67% of its time in that method. Such a renderer can only achieve 2 or 3 frames per second with the Stanford Bunny, while performance with a hand-written rasterizer and z-buffer is at least four times faster.

Table 1 shows the frame rates for various models achieved using a renderer that adds Gouraud Shading, in addition to z-buffer. With low triangle counts, profiling shows that our framerate is largely determined by the cost of transferring the image buffer to the graphics card. Of course, as the triangle count increases, the effect of per-triangle costs increases, and the frame rate drops until we are no longer at interactive speeds.

6. Assignments

Our first course in 3D Computer Graphics is offered as CSCI 391, targeted at third and fourth year students. In many cases, this course is the first truly implementation intensive course that students have taken. The assignments build on each other, and students are encouraged to use their own implementation of previous assignments as a starting point for the next assignment. However, a correct implementation of the previous assignment is always available, preventing a "snowballing" effect when a student runs into trouble. In many cases, a certain amount of "starter code" is provided, to reduce time students spend on issues ancillary to the graphics assignment. When necessary, class files are distributed instead of java files in order to hide an implementation.

A1: *Transformation Matrices.* Students are given code for a renderer that uses Java's built in polygon drawing routines to render tetrahedra in wireframe with a simple parallel projection. Students are asked to implement rotation, scaling, and translation matrices in object space and apply them to vertex lists in order to make the tetrahedra grow, shrink, glide, and rotate.

Figure 8: *A polygonal model of the Utah teapot rendered using Phong Shading.*

A2: *Shapes.* Students write code to generate simple shapes such as cubes, cylinders, cones, and spheres in object space.

A3: *World Space.* Students are asked to implement the WorldSpace class, which keeps track of shape placement in the scene. They must also support rotation and translation in the world coordinate system by associating a new set of matrices with each object in the scene.

A4: *View Space.* The ViewSpace class implements back-face culling, painter's algorithm, and flat shading.

A5: *Perspective and Cameras* Students must implement perspective projection in the Screen Space class, and also provide code that allows the camera to move under mouse control.

A6. *Triangle Rasterization.* Java's built in polygon drawing is discarded in favor of a hand written rasterizer, using the *DDA* implementation.

A7. *Z-Buffer* The rasterizer is augmented to include z-buffer.

A8. *Gouraud Shading* View Space is augmented to compute vertex normals and colors using the Gouraud Shading Model [Gou71].

A9. *Clipping.* Polygon clipping is implemented. The authors have used Cohen-Sutherland [NS79] clipping in Screen Space, but other choices may be explored.

If time permits, one or two additional assignments on Phong shading [Pho75](see Figure 8), texture and bump mapping are possible. Assignment topics are, of course, supplemented with lecture material on radiosity, ray tracing, and other classic topics.

7. Final Remarks

We believe that using Java in a bottom-up approach to teaching Computer Graphics can be successful because we use can Java Programming to teach Graphics, and Graphics to teach Java Programming. Using Java as the implementation language removes some of the pain of implementing large systems in C/C++, allowing students to concentrate more on course content. However, we are teaching important issues in software design, implementation, and analysis of software performance using Graphics as a motivating application.

Although building an animated graphics system in Java may seem an odd choice at first glance, students relate well to *frame rates*, and appreciate seeing the effect that model size and algorithmic differences have on performance. The Easel code base is not only visible to the students, but changing and evolving as the course progresses. We hope that students leave with the sense that they themselves can build anything they like, and can change their environment at will, instead of being restricted to the capabilities of a particular API.

8. Acknowledgements

Part of this work was supported by the National Science Foundation under grant CCF-0541239.

References

[ACSS06] ANGEL E., CUNNINGHAM S., SHIRLEY P., SUNG K.: Teaching computer graphics without raster-level algorithms. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education* (New York, NY, USA, 2006), ACM, pp. 266–267.

[Ang08] ANGEL E.: *Interactive Computer Graphics: A Top-Down Approach with OpenGl (5th Edition)*. Addison-Wesley, 2008.

[Cun98a] CUNNINGHAM S.: An Evolving Approach to Computer Graphics Courses in Computer Science. In *Proceedings of GraphiCon 98* (September 1998).

[Cun98b] CUNNINGHAM S.: Outside the box: the changing shape of the computing world. *ACM SIGCSE Bulletin 30*, 4 (1998), 4–7.

[Gou71] GOURAUD H.: Continuous shading of curved surfaces. *IEEE Transactions on Computers 20*, 6 (1971), 623–628.

[NS79] NEWMAN W., SPROULL R.: *Principles of interactive computer graphics*. McGraw-Hill New York, 1979.

[Pho75] PHONG B.: Illumination for computer generated pictures. *Communications of the ACM 18*, 6 (1975), 311–317.

[SS03] SUNG K., SHIRLEY P.: A top-down approach to teaching introductory computer graphics. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Educators Program* (New York, NY, USA, 2003), ACM, pp. 1–4.

[Sun] SUN MICROSYSTEMS: Tuning garbage collection with the 5.0 java virtual machine. http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html.