Teaching a Modern Graphics Pipeline Using a Shader-based Software Renderer

Heinrich Fink[1], Thomas Weber[1] and Michael Wimmer[1]

[1]Vienna University of Technology, Austria

Abstract

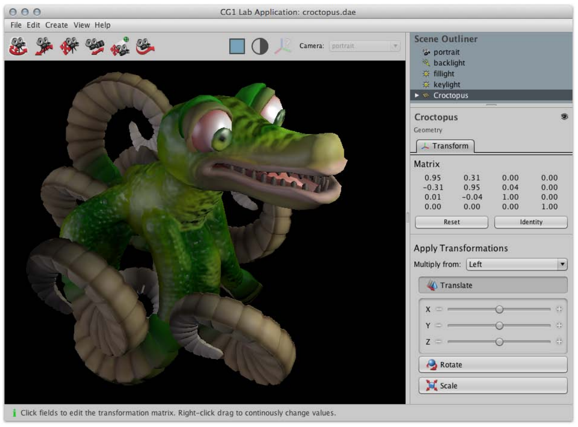
Shaders are a fundamental pattern of the modern graphics pipeline. This paper presents a syllabus for an introductory computer graphics course that emphasizes the use of programmable shaders while teaching raster-level algorithms at the same time. We describe a Java-based framework that is used for programming assignments in this course. This framework implements a shader-enabled software renderer and an interactive 3D editor. We also show how to create attractive course materials by using COLLADA, an open standard for 3D content exchange.

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

1. Introduction

The aim of this paper is to present a new framework for the introductory computer graphics course that we have introduced at the Vienna University of Technology in 2010. We will start by describing related work and the background of this course with our motivation of building a new course framework. We are then going to explain our approach to teach fundamental aspects of a modern graphics pipeline using the concept of shaders. We present our course framework, a Java-based 3D editor with a software renderer and describe how this editor motivates our students during the course by interacting with their work. We also describe how we use COLLADA, an open format for 3D authoring, within our framework and how we solve difficulties in organization and maintenance of the course system. We conclude with our experiences of the new course framework and suggestions for future work. The main contributions of this paper can be summarized as:

- A syllabus for teaching fundamental aspects of a modern graphics pipeline using shaders
- A course framework with a Java-based didactic software renderer and an interactive 3D editor
- Using real-world COLLADA assets to motivate and engage students during the course assignments
- Solutions for overcoming difficulties in course organization and maintenance



0.95	0.31	0.00	0.00
-0.31	0.95	0.04	0.00
0.01	-0.04	1.00	0.00
0.00	0.00	0.00	1.00

Click fields to edit the transformation matrix. Right-click drag to continously change values.

Figure 1: *The editor of the course framework. This scene shows a model that has been exported from the game SPORE by Maxis using the COLLADA format. The matrix display is updated while using the widget in the transform panel.*

1.1. Related Work

The introduction of computer graphics into computer science curricula was first discussed in the late 1980s [Ohl86]. At that time only the most fundamental graphics algorithms such as line drawing and clipping were taught on expensive equipment with highly specialized software [Wol00].

DOI: 10.2312/conf/EG2012/education/073-080

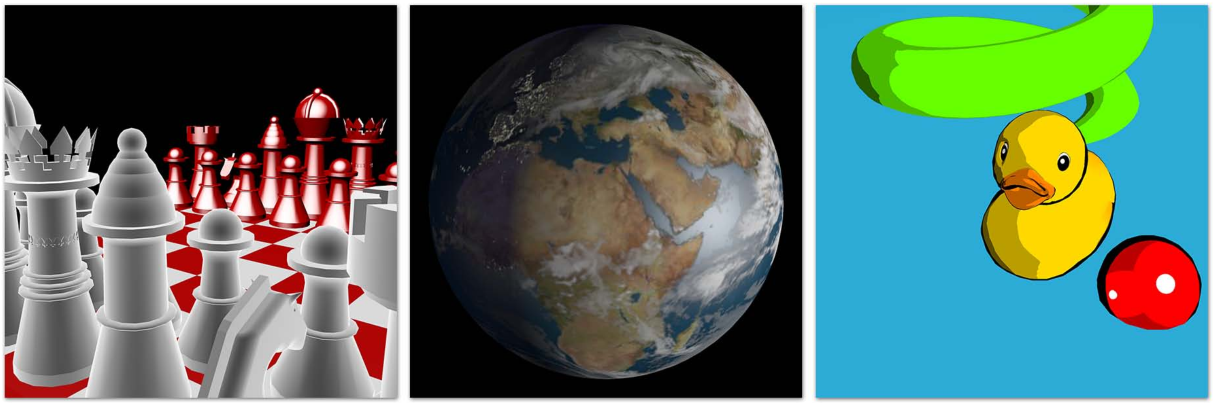


Figure 2: *Entries submitted by students for the annual lab course competition. From left to right: A COLLADA-imported chess scene with custom shaders applied by Philipp Seeböck; Day-and-night shader implementation by Levin Pölser; Cel-shading by Sascha Wiplinger.*

When graphics hardware became available as more affordable mainstream consumer products in the 1990s, most universities started to offer computer graphics courses in a computer science curriculum [Hit00]. At the same time graphics hardware APIs such as OpenGL became widely available. Users of such APIs didn't have to deal with low-level drawing routines anymore. Due to this development, several educators therefore proposed to replace the traditional syllabus of using raster-level algorithms with more practical approaches using higher-level APIs [Cun00, PG99].

Educators further agreed that teaching introductory computer graphics is inherently about 3D geometry, its visual appearance and interplay with lighting simulation and should be taught as interactive projects [HCGW99]. Consequently some courses strongly based their syllabi on scene graph concepts and many introduced Java3D, a Java-based scene graph API that became popular in the late 1990s and early 2000s, to their exercises [PG99, Bou02]. A discussion of how introductory computer graphics courses could benefit from teaching scene graphs is given by Cunningham et al. [CB01].

At the 2004 SIGGRAPH education workshop [CHLS04], the prevalent opinion was that introductory computer graphics courses should be made available to every undergraduate computer science student and not just to those who specialize in this field during their studies. For students with a less traditional background it seemed more appropriate to teach the higher level modules of a graphics application first (*top-down*) as opposed to traditionally beginning the course with raster-level operations and gradually moving towards higher level concepts (*bottom-up*). The *top-down* approach showed to work well for more mature students who took only a single computer graphics course during their studies [SS04]. However, the discussions of the 2006 SIGCSE panel [ACSS06] suggest that there is no right way to build a computer graphics curriculum, and that teaching the *bottom-up* approach as an introductory course would still be a viable approach for those students who follow up with a series of advanced computer graphics courses.

While previous discussions mainly focused on the structure and content of the syllabus, more recent discussions emphasize the importance of the context in which computer graphics are being taught [CC09, Cun08]. Choosing a context which allows students to work on problems that are also relevant outside the course environment turned out to highly boost motivation. For example, Schweppe et al. used the context of theatre [SG09] for teaching computer graphics.

With the wide availability of programmable graphics hardware, approaches of teaching shaders in computer graphics courses have been increasingly investigated [OZCP05, BC07, TF07]. As shaders became commonly used in graphics programming, a shader-based introductory computer graphics course was recently proposed [AS11].

2. Course Background and Motivation

At Vienna UT, the curriculum *Visual Computing* offers three main courses focused on computer graphics:

1. An introductory course teaching fundamental aspects of computer graphics using Java.
2. An intermediate course teaching modern OpenGL with C++.
3. An advanced course on state-of-the-art optimization methods and graphics effects.

Each course has lectures covering theoretical aspects that are applied in lab exercises with programming assignments. Other advanced electives with related topics such as visualization, virtual and augmented reality or color sciences are also offered regularly. The outline of the *Visual Computing* curriculum is largely based on the ACM recommendations [REC*01].

	Topics	Interaction w. Shader Concepts
Assignment 1	Bresenham line rasterization Viewport mapping	
Assignment 2	3D vector/math operations Model transformations	Execute vertex shading stage Apply model matrix to vertices in vertex shader (VS)
Assignment 3	Polygon clipping View and projection transform Linear color interpolation	Interpolate per-vertex attributes for clipped polygons Concatenate model-view-projection (MVP) matrix Pass MVP matrix and view matrix to VS Apply MVP matrix to vertices in VS Add interpolation of varyings to line rasterizer Pass vertex color from VS to fragment shader (FS) as varyings Return interpolated color in FS
Assignment 4	Triangle fill rasterization Back-face culling Depth test with Z-buffer	Interpolate varyings with barycentric coordinates Call FS with interpolated varyings
Assignment 5	Transforming normals Shading models Illumination models	Calculate the inverse-transpose of the model matrix Pass inverse-transpose to VS Create shaders for per-vertex and per-fragment lighting Transform normal, view and light vector to world space using the VS Calculate Lambert/Blinn-Phong illumination in world space in the FS
Assignment 6	Texturing Perspective-correct interpolation	Use *sampler* uniforms in FS Pass UV coordinates as varyings between VS and FS Freely experiment with new custom shaders

Table 1: *Overview of our syllabus and how we gradually approach shaders with each graphics topic.*

In the European Space of Higher Education (ESHE), curricula have to be split into two cycles in accordance with the bologna requirements [FPAA06]. At Vienna UT, the introductory and intermediate computer graphics courses are compulsory during the first education cycle (the *Visual Computing* bachelor program). These courses teach the theoretical foundations that are necessary to continue with advanced degrees, as well as practical skills such as modern OpenGL with C++ that are often required for practical work in the computer industry. This is in line with recommendations made during a previous education workshop where the consequences of the Bologna process for computer graphics education have been discussed [BCFH06].

This paper describes the first computer graphics course which is usually taken by second-year full-time computer science students. The prerequisites for this course are basic programming skills in Java, object-oriented modeling and basic linear algebra. In the *Visual Computing* curriculum, every student has had courses covering these subjects before attending the introductory computer graphics course. Our course is attended by approximately two hundred students each year. This comparatively high number of students poses challenges in distribution and maintenance of course material. We address these issues in Section 4.3.

For the introductory course we chose the *bottom-up* approach where a large amount of time is spent on implementing fundamental graphics operations such as triangle rasterization, viewing and visibility algorithms. As argued by Shirley [ACSS06], we believe that this approach is more effective in communicating the basic computer graphics algorithms. The *bottom-up* approach has also been considered [SS04] to work better for courses like ours where the majority of students is enrolled full-time and where more than one computer graphics course is offered.

However, it was also our motivation to teach concepts that are practically relevant. While teaching the second, intermediate graphics course, we experienced that many students had problems adopting the modern approach of shader-based OpenGL. Shaders are programming patterns that are now mandatory in any recent real-world graphics API [AS11] (OpenGL 3.2+, OpenGL ES 2.0, WebGL, etc.). Therefore they are highly relevant to graphics programming. We decided that the concept of shaders should form a fundamental part of our syllabus and that they should be included while teaching the more traditional lower-level algorithms.

3. Course Syllabus

The aim of the introductory laboratory course is for the students to apply the concepts described in the lecture. These topics include raster-level algorithms, polygon clipping, 3D transformations, hidden surface removal, lighting models and texturing.

At the beginning of each course, students receive an incomplete version of a software renderer. We have defined six programming assignments that incrementally add features to the graphics pipeline. Students are supposed to solve these assignments individually.

A summary of the assignments and how we include the concept of shaders to our syllabus is found in Table 1. We would also like to refer to the online Wiki of the lab course [CGLb] that describes the assignments in detail.

What distinguishes our approach from other courses is that it communicates aspects of a modern graphics API while still being software-based: instead of a fixed-function pipeline with a handful of illumination models and shading modes, a fully programmable shader-based approach is used. This is motivated by the fact that any current graphics API requires the use of a vertex and fragment shader [AS11].

Shaders are implemented as classes and interact with the rasterizer pipeline through polymorphism. The classes implement an interface which defines entry points of the vertex and fragment shader as abstract methods. Students will implement both the shader classes and parts of the renderer that interacts with them. This way students learn about shader-based computer graphics from day one and also get a chance to see how shaders are employed within a graphics pipeline. This is not possible with current graphics APIs because these parts are usually hidden from the programmer.

The typical shader inputs and outputs (vertex attributes, uniform values, varying values and the final fragment color) are all included in the software model. Vertex attributes are a fixed set of per-vertex values: color, normal, tangent, bitangent and uv coordinates. Uniforms are represented by member variables of the shader object. The fragment output is a single RGB color triple. Varying values, which are the output of the vertex-shader, are interpolated during rasterization and then become the input of the fragment-shader. These are encapsulated in a specialized class which consists of an array of float values and methods for interpolating them. Perspective-correct interpolation is also supported.

The course is split into six assignments which build upon each other. This allows students to better understand the big picture as opposed to isolated assignments that solve only one particular problem. While we supply a standard solution after each finished task, many students choose to use their own solutions from start to end.

The first assignment is a straightforward and simple task: Students implement Bresenham line rasterization and complete the viewport transform of points from normalized device coordinates to pixel coordinates. This gives them time to set up the development environment and get accustomed to the framework.

In the second assignment, students implement model transformations and general 3D math operations like the dot-product and matrix-multiplication. This is also the first time they use shaders, when calling the vertex-shader and applying the model-matrix to the input vertices. After completing this task, students should understand the concept of 3D transformations, be able to build the inverse for combinations of common transformations and understand the difference between matrix-multiplication from the right and from the left.

```
public class LambertGouraudShader extends SurfaceShader {

  @RGBParam(r = 1, g = 1, b = 1)
  public Vec3 diffuse;

  @Override
  public Vertex shadeVertex(Mesh.Vertex v) {

    Vec4 pos = Mat4.mul(_modelViewProjectionMatrix,
                        v.position);

    Vec3 P = Mat4.mul(_modelMatrix,
                      v.position).homogenize3();

    Vec3 N = Mat3.mul(_normalMatrix,
                      v.normal).normalize();

    Vec3 C = v.color;
    Vec3 surfaceColor = Vec3.mul(diffuse, C);

    Vec3 I = IlluminationModels.lambert(P,
                                        N,
                                        surfaceColor,
                                        _lights);

    Varyings vr = new Varyings(new float[]{I.r(),
                                           I.g(),
                                           I.b()});

    return new Vertex(pos, vr);
  }

  @Override
  public Vec3 shadeFragment(Varyings varyings) {

    float values[] = varyings.getValues();

    return new Vec3(values[0], values[1], values[2]);
  }
}
```

Listing 1: *Shader implementation of Gouraud-shading with Lambert-illumation*

In the third assignment, students implement polygon-clipping in homogeneous coordinates and complete the viewing pipeline by adding viewing and projection matrices. They also implement the necessary sections to interpolate per-vertex colors using the vertex and fragment shaders. Upon completion of this task, students have learned about the complete 3D viewing-pipeline and polygon clipping.

The topic of the fourth assignment is triangle rasterization. Students implement a triangle rasterizer based on evaluating plane equations. We chose this simple algorithm over commonly used algorithms like scan-line filling, because this is similar to how modern graphics hardware implement rasterization. Varying vertex shader outputs are interpolated using barycentric coordinates and passed to the fragment shader. Hidden surface removal using depth testing and back-face culling is also implemented in this task.

When reaching assignment five, students have completed a simple, yet flexible rendering system. At this point their task is to implement different types of lighting using shaders. Two types of shading (Gouraud and Phong) as well as two

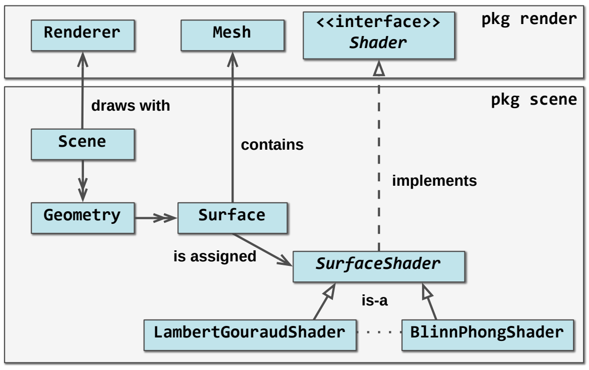


Figure 3: *A simplified relationship diagram that shows how classes from the scene package use public interfaces from the render package.*

illumination models (Lambert and Blinn-Phong) have to be implemented.

In the final assignment, students use textures and implement a shader effect of their choice. They also add perspective-correct interpolation of varyings. This is necessary to correctly pass UV coordinates between vertex and fragment shading. The remaining time of this task is an open assignment to encourage experimenting and to explore the potential of shaders. We provide a list of examples and suggestions (e.g. alternate lighting models, normal mapping, etc...) to assist students in finding a topic.

Students can test and interact with their solution through a simple 3D editor that uses their solutions as the renderer backend. For each submission, starting with the second assignment, students use this application to create example scenes. These scenes showcase implemented features of the particular task. For some assignments we also ask students to show things like the difference between left- and right-multiplication in assignment two, or three-point-lighting in assignment five. The target application is explained in more detail in the next section.

4. Course Framework

We have built a custom course framework in Java that implements the concepts described by the previous section. We have chosen Java as the programming language for this course, because it is taught as the introductory programming language in the first year of our curriculum. Java has been used many times as the language of choice for introductory graphics courses [Muk99, TJN06, RY09]. We agree that garbage collection, boundary checks of array access and simple debugging facilities help students to focus on more relevant aspects of their implementation.

The source code of our course framework is organized in different modules as Java packages. During each assignment students have to complete sections of code in one or more of the following packages:

- **`render`**: this package contains the implementation of our graphics pipeline model as described by the previous section. The public interfaces in this package correspond to the *API layer* of a modern graphics API that is visible to the application code.
- **`scene`**: classes in this package implement a simple data model of a scene that consists of geometries, cameras and light sources. A scene uses the render package to store render data and to draw itself (see Figure 3). User implementations of shaders are also included in the scene package, such as the shader showed in Listing 1. Code contained in this module largely represents the *client code* of a graphics API, i.e. code that uses a graphics hardware API to draw a 3D scene.
- **`math`**: a collection of classes and methods for linear algebra routines with vectors and matrices.

For a complete description of classes we refer to the online JavaDoc documentation of our framework [CGLa].

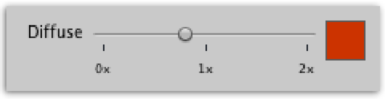
4.1. The target application: a simple 3D Editor

In order for the students to interact with their solution, we have added a simple and easy-to-use 3D Editor to our framework. This editor is written solely in Java and uses Java Swing for displaying a graphical user interface (GUI). A recent version of the application is publicly available online [CGLc].

The application (see Figure 1) creates, loads and saves 3D scenes. The status message in the lower left corner provides useful context information and tool-tips. This helps students to quickly understand the features of the application.

Geometry, light and camera objects that are contained in a scene are accessible through the scene outliner in the upper right corner. These objects can be selected, added or deleted. Below the scene outliner another panel shows the properties of the selected object. This panel should motivate students to play around with parameters of the framework and their solution. The position and orientation of this object can be modified by applying a translation, rotation or scaling through a GUI widget. Alternatively, the values of the transformation matrix can be entered directly. The display of this matrix is updated interactively each time the object's transformation has changed (see Figure 1). Any float parameter in the properties panel can also be modified continuously by dragging the mouse. The render view to the left is updated in real time and immediately reflects a parameter change.

Uniforms of shader instances are also editable in the properties panel. Classes that implement shaders use Java Annotations to mark class members as uniforms and to provide the GUI with additional information. For example, the `diffuse` parameter in Listing 1 is annotated by `RGBParam` which defines a default color. The GUI then automatically adds the following panel to the properties display:



Introspection is often used in the implementation of the framework and ultimately allows students to quickly interact with the uniforms of their own shaders in the GUI through automatically generated widgets.

There are multiple ways of navigating within a scene: tumble mode, dollying, zooming, trackball rotation and walkthrough mode. While navigating, the transformation matrix of the active render camera is modified and its matrix display is updated in real time. The GUI thread is asynchronous to the renderer thread of the scene. This results in a very responsive GUI. Most scenes that are used during the course render at 30fps or higher on current laptops. In our opinion, a good user experience with the GUI encourages students to playfully explore the topics of the course.

The GUI and content pipeline that is described in the following consist of a considerable amount of source code. However, the six programming assignments of this lab course are completely independent from their implementation. We have therefore packaged their classes into a separate Java Archive (JAR). This hides complex code that might distract students. It also allows us to easily distribute bug-fixes of the GUI by posting new versions of this JAR file during the semester.

4.2. Employing a COLLADA-based content pipeline

We aim to provide our students with good-looking and interesting content for the editor. COLLADA is an open industry-standard XML format for exchanging 3D content [col]. It is maintained by the Khronos Group which is also organizing the OpenGL graphics API standard process. We chose COLLADA as the primary data format of our framework. Many popular 3D applications have importers and exporters available. These include Autodesk Maya, Blender, Google Sketchup and even games like Maxis SPORE (see Figure 1). Our application saves, loads and imports COLLADA scenes directly. This allows us to access an enormous amount of online 3D assets. Google's 3D Warehouse [goo], for example, hosts thousands of free COLLADA scenes that can be opened and rendered by our framework.

While the largest part of our scene data model is a subset of the COLLADA standard, scene attributes that are special to our framework are stored as extra elements with COLLADA's extension mechanism. This doesn't break the validity of a COLLADA file. Any scene that is saved by our framework, can still be opened by any COLLADA-compatible application. We also believe that students might benefit from the human-readable XML format by looking at the elements that compose a scene.

At the end of each course we organize a competition that engages students to build interesting scenes with the editor and to experiment with custom shaders (see Figure 2). Importing COLLADA models enables them to incorporate real-world 3D assets either by downloading online content or by importing assets from other 3D applications.

4.3. Deployment of assignments

After the deadline of an assignment has passed, we provide the students with a version of the framework that has the previous assignments completed while still missing the features of the upcoming ones. In order to avoid maintaining multiple source trees, we have created a markup system to tag those sections of code that we expect our students to implement (see Listing 2).

```
 1 // Iterate over all vertices
 2 for (int i = 0; i < mesh.getVertexCount(); ++i) {
 3
 4     Mesh.Vertex meshVertex = mesh.getVertex(i);
 5
 6     //#task 2 "Execute vertex shader stage"
 7     // Transform and shade all vertices
 8     Vertex v = shader.shadeVertex(meshVertex);
 9     vertices[i] = v;
10
11     //#spec
12     /**
13      * TODO 2:
14      * - Transform the vertices by calling the vertex
15      *   shader.
16      * - You can access a vertex of a mesh by calling
17      *   mesh.getVertex.
18      */
19 //  // Delete me
20 //  vertices[i] = new Vertex(meshVertex.position,
21 //                           Varyings.empty);
22     //#endtask
23 }
```

Listing 2: *Excerpt from the Java class `Renderer`. The tag `#task` in line 6 indicates that code from there until the `#spec` tag contains the reference solution for a code snipped of assignment 2. Between line 11 (`#spec`) and 22 (`#endtask`) we can see the code that is presented to the student in the beginning. Line 19 to 21 would automatically be uncommented by our parser that generates the student's version.*

The main source tree which is maintained and continuously developed always compiles the full reference solution. When we build a student version of the framework, a Python script parses the previously shown `#task` annotations and - depending on the number of the assignment - automatically strips the reference solution, replaces it with the comments describing the task and placeholder code.

This process allows us to automatically derive a variety of resources from a single source branch, such as:

- Sources of reference solutions for each assignment;
- Prebuilt executable JARs of each reference solution;

- Internal Wiki pages showing code segments for each assignment;
- Reference renderings of solutions for each task;
- Web resources such as Java Webstart wrappers, download packages, and many more.

Our course is attended by approximately two hundred students per year. Each student has to discuss the solved assignments with a member of the faculty for evaluation of grades. More than 15 members of the faculty help out to hold these evaluations. Documentation that is updated regularly helps to organize this process more efficiently.

5. Discussion

Rhodes et al. implemented a similar course with EASEL, a didactic software-based rasterizer in Java [RY09]. They recommend to avoid small allocations on the heap in favor of allocating larger blocks and to re-use objects whenever possible. In our design we have decided to favor code clarity and modularization over code-level optimizations. In many situations we trust Java's HotSpot optimizing compiler to avoid potential performance impacts because of a higher-level class design. One feature of this just-in-time compiler is *escape analysis* [CGS*99] where local allocations are optimized to stack memory. Escape analysis is enabled by default in Java SE 6u23 and later. The shader shown in Listing 1, for example, allocates new instances of `Mat4` classes for each matrix multiplication. This shader renders the Stanford Bunny consisting of 69451 triangles with approximately 11 fps on a Core 2 Duo processor at 2.53 Ghz using single-threaded rasterization. The EASEL framework reports a performance of 11.3 fps with a slightly faster processor [RY09]. This suggests that our choice of using easier-to-read and small classes does not result in a significant impact on performance.

The framework has been used in two iterations of the course. In the first iteration of the new framework we have experienced that students tend to implement the code snippets only by following the instructions without actually spending time experimenting with the topic at hand. Our dynamic markup system as described in section 4.3, allowed us to quickly change the assignments and to adapt to this problem for the second iteration. We have changed later assignments to be more open (e.g. by adding custom shaders).

The content competition held each semester turned out to be very successful and we believe that we could also motivate those students who had no previous experience with computer graphics to participate. Figure 2 shows submissions by students for the competition.

An anonymous evaluation by students held each semester showed above average ratings of our course. We have also received very positive feedback from students during the discussions of the assignments. However, the framework is still very young and while we think that our students are better prepared for our advanced computer graphics courses, it is too early to actually measure the impact of the new framework.

We think that our framework covers a broad range of fundamental topics in computer graphics. There are, however, features that we haven't implemented. Hierarchical transformations have been shown to be valuable to an introductory computer graphics course [CB01]. We have decided to leave out this feature in favor of simpler data structures in code used by students. We also don't support alpha blending in the framebuffer. This limits the framework to opaque materials only. We use multiple threads for each primitive during the rasterization stage to reach interactive frame rates on recent multi-core CPUs. We think that this approach is not optimal and that we should rather implement true pipelining of each stage in the renderer. This would also provide our students with a more useful lesson in parallel programming.

6. Conclusion

We presented a syllabus that teaches the concept of shaders while still employing raster-level algorithms of the graphics pipeline. Our framework implements this concept effectively and has shown to be successful in providing students with an interactive learning environment. The integration of the COLLADA format proved to increase the quality of our course materials and to heighten the motivation of our students. We believe that our students are well prepared for advanced courses in computer graphics after completing this course.

In the future we would like to use our framework for demonstration purposes during the lecture of this course, and we would like to evaluate its use for other related courses. We will also submit the complete course framework to the CGEMS [CGE] repository in order to share our resources with other educators.

7. Acknowledgments

We would like to thank our admin Stephan Plepelits for creating the web-based submission system for this course and for always helping us out with last-minute requests. We would also like to thank our students who constantly give us feedback to improve the course and its framework. For the second iteration of our course, Dominik Rauch, a former student, has added texturing to our software renderer which greatly improved the possibilities of our framework. Finally, we would like to thank the head of our group, Prof. Werner Purgathofer for trusting us with the laboratory course of his introductory computer graphics lecture and for encouraging us in our work.

References

[ACSS06] ANGEL E., CUNNINGHAM S., SHIRLEY P., SUNG K.: Teaching computer graphics without raster-level algorithms. *ACM SIGCSE Bulletin 38*, 1 (Mar. 2006), 266. 2, 3

[AS11] ANGEL E., SHREINER D.: Teaching a Shader-Based Introduction to Computer Graphics. *Computer Graphics and Applications, IEEE 31*, 2 (2011), 9–13. 2, 3, 4

[BC07] BAILEY M., CUNNINGHAM S.: A hands-on environment for teaching GPU programming. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education* (Mar. 2007), ACM Request Permissions. 2

[BCFH06] BOURDIN J., CUNNINGHAM S., FAIRÉN M., HANSMANN W.: Report of the cge 06 computer graphics education workshop. *Vienna, Austria* (2006). 3

[Bou02] BOUVIER D.: From pixels to scene graphs in introductory computer graphics courses. *Computers & Graphics 26*, 4 (2002), 603–608. 2

[CB01] CUNNINGHAM S., BAILEY M. J.: Lessons from scene graphs: using scene graphs to teach hierarchical modeling. *Computers & Graphics 25*, 4 (Aug. 2001), 703–711. 2, 7

[CC09] CASE C., CUNNINGHAM S.: Teaching computer graphics in context. *Computer Graphics Education 9* (2009), 29–30. 2

[CGE] Cgems educational resources. `http://cgems.inesc.pt`. 7

[CGLa] Course framework javadoc documentation. `https://lva.cg.tuwien.ac.at/cgl/javadoc`. 5

[CGLb] Lab course wiki documentation, available online: `https://lva.cg.tuwien.ac.at/cgl/wiki`. 4

[CGLc] Webstart launcher of the 3d editor, available online: `https://lva.cg.tuwien.ac.at/cgl/go`. 5

[CGS*99] CHOI J.-D., GUPTA M., SERRANO M., SREEDHAR V. C., MIDKIFF S.: Escape analysis for Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Oct. 1999), ACM Request Permissions. 7

[CHLS04] CUNNINGHAM S., HANSMANN W., LAXER C., SHI J.: The beginning computer graphics course in computer science. *SIGGRAPH Computer Graphics 38*, 4 (2004). 2

[col] Collada: a standard for digital content authoring. `http://www.khronos.org/collada/`. 6

[Cun00] CUNNINGHAM S.: Powers of 10: the case for changing the first course in computer graphics. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education* (Mar. 2000), ACM Request Permissions. 2

[Cun08] CUNNINGHAM S.: Computer graphics in context: an approach to a first course in computer graphics. *ACM SIGGRAPH ASIA 2008 educators programme* (2008), 1. 2

[FPAA06] FULLER U., PEARS A., AMILLO J., AVRAM C.: A computing perspective on the Bologna process. *ACM SIGCSE Bulletin* (2006). 3

[goo] Google warehouse. `http://sketchup.google.com/3dwarehouse/`. 6

[HCGW99] HITCHNER L., CUNNINGHAM S., GRISSOM S., WOLFE R.: Computer graphics: the introductory course grows up. In *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education* (Mar. 1999), ACM Request Permissions. 2

[Hit00] HITCHNER L.: Adapting computer graphics curricula to changes in graphics. *Computers & Graphics 24*, 2 (Apr. 2000), 283–288. 2

[Muk99] MUKUNDAN R.: Teaching computer graphics using Java. *ITiCSE-WGR '99: Working group reports from ITiCSE on Innovation and technology in computer science education* (1999). 5

[Ohl86] OHLSON M. R.: The role and position of graphics in computer science education. In *SIGCSE '86: Proceedings of the seventeenth SIGCSE technical symposium on Computer science education* (Feb. 1986), ACM Request Permissions. 1

[OZCP05] OWEN G. S., ZHU Y., CHASTINE J., PAYNE B. R.: Teaching programmable shaders: lightweight versus heavyweight approach. *SIGGRAPH '05: SIGGRAPH 2005 Educators program* (July 2005). 2

[PG99] PETER I., GUMHOLD S.: Teaching computer graphics with java 3d. *Advances in Multimedia and Distance Education (Proceedings of ISIMADE'99)* (1999). 2

[REC*01] ROBERTS E., ENGEL G., CHANG C., CROSS J., SHACKELFORD R., SLOAN R., CARVER D., ECKHOUSE R., KING W., LAU F.: Computing Curricula 2001: Computer Science. *Los Angeles/New York: IEEE Computer Society/Association for Computing Machinery [URL: http://www. acm. org/sigcse/cc2001/cc2001. pdf]* (2001). 2

[RY09] RHODES P. J., YAN B.: Easel: A Java Based Top-Down Approach to 3D Graphics Education. In *EG 2008 - Eduaction Papers* (Dec. 2009), pp. 29–36. 5, 7

[SG09] SCHWEPPE M. K., GEIGEL J.: Teaching Computer Graphics in the Context of Theatre. In *EG 2009 - Education Papers* (2009), pp. 67–72. 2

[SS04] SUNG K., SHIRLEY P.: A top-down approach to teaching introductory computer graphics. *Computers & Graphics 28*, 3 (2004), 383–391. 2, 3

[TF07] TALTON J. O., FITZPATRICK D.: Teaching graphics with the openGL shading language. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education* (Mar. 2007), ACM Request Permissions. 2

[TJN06] TORI R., JOÃO JR, NAKAMURA R.: Teaching introductory computer graphics using java 3D, games and customized software: a Brazilian experience. *SIGGRAPH '06: SIGGRAPH 2006 Educators program* (2006). 5

[Wol00] WOLFE R.: Bringing the introductory computer graphics course into the 21st century. *Computers & Graphics 24*, 1 (2000), 151–155. 1