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

H. Fink[a], T. Weber[a], M. Wimmer[a]

[a] *Vienna University of Technology, Austria*

Abstract

This paper presents the 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. Teaching shader programming in concert with the low-level graphics pipeline makes it easier for our students to learn modern OpenGL with shaders in our follow-up intermediate course. We also show how to create attractive course material by using COLLADA, an open standard for 3D content exchange, and our approach to organizing the practical course.

Keywords:
CG education, course organization, COLLADA, programmable shading

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 (Fig. 1) 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 exchanging digital 3D assets, 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

This paper is based on an earlier conference paper [1]. We extend the discussion and give more details on the specifics of our software framework and course organization.

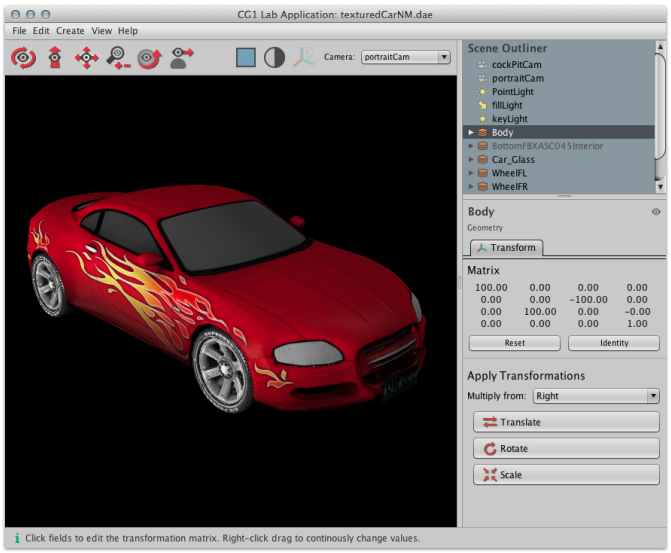

Figure 1: The editor of the course framework. This scene shows a model that has been imported using the COLLADA format. The matrix display is updated while using the widget in the transform panel.

1.1. Related work

The introduction of computer graphics into computer science curricula was first discussed in the late 1980s [2]. At that time only the most fundamental graphics algorithms such as line drawing and clipping were taught on expensive equipment with highly specialized software [3]. 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 [4]. At the same time graphics hardware APIs such as OpenGL became widely available. Users of such APIs did not have to

Email addresses: hfink@cg.tuwien.ac.at (H. Fink), t.weber@cg.tuwien.ac.at (T. Weber), wimmer@cg.tuwien.ac.at (M. Wimmer)

deal with low-level drawing routines anymore. Due to this development, several educators proposed to replace the traditional syllabus of using raster-level algorithms with more practical approaches using higher-level APIs [5, 6].

During the thirtieth SIGCSE technical symposium on Computer science education, educators 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 [7]. 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 [6, 8]. A discussion of how introductory computer graphics courses could benefit from teaching scene graphs is given by Cunningham and Bailey [9].

At the 2004 SIGGRAPH/Eurographics education workshop [10], 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 [11]. However, the discussions of the 2006 SIGCSE panel [12] 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 [13]. 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 and Geigel used the context of theatre [14] for teaching computer graphics.

With the wide availability of programmable graphics hardware, approaches of teaching shaders in computer graphics courses have been increasingly investigated [15, 16, 17]. As shaders have become commonly used in graphics programming, a shader-based introductory computer graphics course has recently been proposed [18].

2. Course background and motivation

At the Vienna University of Technology (VUT), the bachelor program *Media Informatics and Visual Computing* and the master program *Visual Computing* offer three main courses focusing 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 graphics effects and optimization methods.

Each course has lectures covering theoretical aspects that are applied in a practical part 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 our curriculum is largely based on the ACM recommendations [19].

In the European Space of Higher Education (ESHE), curricula have to be split into two cycles in accordance with the Bologna requirements [20]. At VUT, the introductory and intermediate computer graphics courses are compulsory during the first education cycle (the *Media Informatics and 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 were discussed [21].

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 our curriculum, every student has had courses covering these subjects before attending the introductory computer graphics course. In particular, Java is taught as the introductory programming language in the first year of our curriculum and is therefore the programming language of our choice for this course. Our course is attended by approximately 150 students each year. This comparatively high number of students poses challenges in distribution and maintenance of course material. We address these issues in Section 5.

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. The majority of our students is enrolled full-time and is likely to attend the intermediate course later. We believe that it is easier for our students to learn more advanced computer graphics algorithms and higher-level APIs in later courses when having learned the fundamental algorithms first, hence our choice for the *bottom-up* approach. Previous publications on applying the *top-down* approaches to introductory courses support this decision, where a *bottom-up* approach is still considered to work well in traditional computer science curricula [12, 11].

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 [18] (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, texturing and shaders.

At the beginning of the semester, students receive an incomplete version of a software renderer. We 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 in our syllabus is found in Table 1. We would also like to refer to the online Wiki of the lab course[1] that describes the assignments in detail.

What distinguishes our approach from other courses is that it communicates aspects of a programmable graphics pipeline 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 [18].

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. The main thing students are supposed to learn in this exercise is to set individual pixels in a framebuffer. Any line drawing algorithm could be used for this. We chose the DDA algorithm in the first iteration of the course for instance.

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.

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 line equations. Varying vertex shader outputs are interpolated using barycentric coordinates and passed to the fragment shader. Hidden surface removal using depth testing and backface 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 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 implementation of the renderer for live rendering. That way, our students are able to easily find bugs and to explore the behaviour of their code. For each submission, starting with the second assignment, students also 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

Students implement the assignments described in the previous section inside a custom Java framework. Java has been used many times as the language of choice for introductory graphics courses [22, 23, 24]. 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. Using Java also allows us to support any major desktop operating system like Windows, Linux or Mac OS X without having to write platform-specific code.

4.1. Student packages

The framework is organized into several modules as Java packages. During each assignment, students have to complete sections of code in the following three 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

[1] https://lva.cg.tuwien.ac.at/ecg/wiki

	Points	Topics	Interaction w. Shader Concepts
Assignment 1	6	Bresenham line rasterization Viewport mapping	
Assignment 2	8	3D vector/math operations Model transformations	Execute vertex shading stage Apply model matrix to vertices in vertex shader (VS)
Assignment 3	14	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	12	Triangle fill rasterization Back-face culling Depth test with Z-buffer	Interpolate varyings with barycentric coordinates Call FS with interpolated varyings
Assignment 5	12	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	8	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.

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 2). User implementations of shaders are also included in the scene package, such as the shader shown 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.

The division between the `scene` and `render` package is intended to model the separation between user code and graphics driver in a typical 3D application. This is why rasterization, clipping, framebuffers and any code that calls shaders is implemented in the `render` package, but scene management, matrix calculation and the actual shader implementation happens in the `scene` package. Figure 2 shows how these two packages interact with each other.

Since the code of the framework is intended to be read, understood and modified by novice programmers, great care has been taken to use a didactic style when writing them. This means that we generally avoid hard-to-follow concepts like recursion and favor interface implementation over class inheritance if polymorphism is required. Code is kept as simple and compact as possible and is thoroughly commented. A single

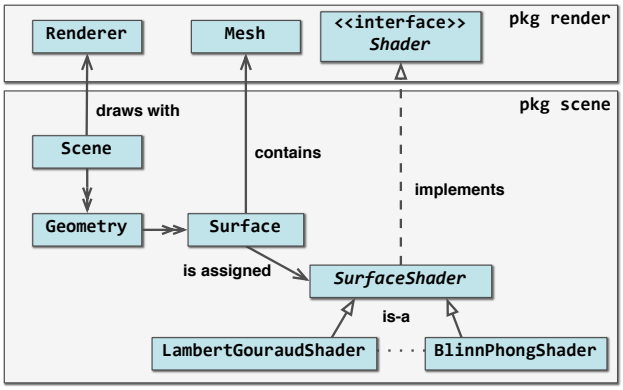

Figure 2: A simplified relationship diagram that shows how classes from the scene package use public interfaces from the render package.

graphics algorithm is usually encapsulated in a single class or method.

Shaders are implemented as classes in the `scene` package 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.

```
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

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.

For a complete description of classes we refer to the *JavaDoc* documentation of our framework, which is available online[2].

4.2. 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 and can be launched directly using Java Webstart[3].

The application (see Figure 1) creates, loads and saves 3D scenes and uses the classes in the student modules for scene management and rendering. 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 a panel as shown in Figure 3 to the properties display.

Figure 3: A color-picking widget that has been automatically created from an annotated shader uniform. By clicking on the colored square, a color can be selected using a standard Swing dialog. The horizontal slider allows setting the intensity of the color. The slider allows boosting the colors until 200 percent (depicted as *2x* by the widget). This is often useful for artistic control of light source parameters.

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 walk-through 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

[2] https://lva.cg.tuwien.ac.at/ecg/javadoc

[3] https://lva.cg.tuwien.ac.at/ecg/go

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 which is described in the following subsection 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.

Images rendered by the students and textures loaded from disk are automatically gamma-corrected. Students do not implement this, but we believe it is important that a student's first experience with rendering and lighting happens in a linear work flow.

4.3. 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 [25]. 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 and other tools like Google Sketchup. 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 [26], 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 does not 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. Importing COLLADA models enables them to incorporate real-world 3D assets either by downloading online content or by importing assets from other 3D applications.

5. Course organization and maintenance

In this section we describe the organization of our course and the solutions we have found for common problems. Our course is attended by approximately 150 students each year. This high number of students makes organizing a practical course that retains a certain level of individuality and personal support for each student difficult. Because of this, we tried to automate many aspects of this process.

Doing so allowed us to support such a large course with relatively little resources. If this same course were attended by a smaller number of students many of the following methods (especially those discussed in Section 5.2, 5.3 and 5.4) wouldn't be necessary. A more personal organization style would be preferable in this case.

5.1. 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).

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 comments that contain the tag `#task` and – depending on the number of the assignment – automatically strips the reference solution, replacing it with the comments describing the task and placeholder code.

```
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 part 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.

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 JAR files 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.

All of this is controlled by a single Apache Ant build script, where all of these are individual targets. The build script also allows preprocessing, building and executing any assignment with a single command.

In case of a necessary code change during an ongoing semester, the build script can be used to generate and upload all necessary content files, reference solution packages, reference renders etc. to our servers with a single command.

Student source packages come with the prebuilt GUI packages, incomplete sources for the student framework and a simple build script that allows building and running the application. The build script is also used to pack submission archives of the student's solution so that it can be uploaded to the submission system. Previously, students would pack these archives themselves, but this was found to be a common source of errors. Students would often pack their submission packages at the wrong folder level, add unnecessary files or forget important files. By automating this and setting up the submission system to reject archives not created in this manner, we can avoid these problems completely.

5.2. Online resources and support

The main source of information for students is the course wiki[4]. Here, they can find a detailed description of each assignment along with any necessary information they might need for implementation. At the end of each assignment description, students can find a list of reference scenes along with images rendered by the reference implementation. Students can download these scenes and check if their solution is behaving correctly. The reference images are automatically generated and uploaded using our Ant build script. In the wiki they also find a general information on the structure of the framework and on how to set up their development environment for the course.

The wiki allows students to write comments in a section below every article, but we found that this feature was seldom used. In the first iteration of the course, students also had the ability to edit articles. We hoped this would encourage students to discuss and edit unclear sections themselves, but this did not happen.

Students who want to submit an image for the annual content competition will do this by creating a new article in the wiki. In order to support students, we provide an online forum where questions are answered usually within one day, and we also hold personal weekly support hours where several tutors would be available in person to discuss more in-depth problems.

5.3. Grading

Students get points for each assignment for a total of 60 points. The number of reachable points per assignment can be found in Table 1. In order to get the full points, they have to upload their submission on time. Every day of over-time leads to 2 points deduction. In addition to that, each student has to discuss the solved assignments with a member of the faculty for evaluation of grades.

During the evaluation talks, the grading faculty member checks the completeness of the solution and asks questions about both the practical and theoretical nature of their implementation. The submitted student scene is also checked and discussed at that point. Afterwards, the evaluator decides the final number of points reached for each assignment.

These talks happen twice per semester and take about 15 minutes. This very short amount of time makes it necessary to enable graders to get the relevant information on a submission at the click of a button. A side-by-side comparison between the render output of the student's solution and the output of the reference implementation provides a good first look at the correctness of the student's solution. Automatically generated weblinks to the changed source parts of the student's solution and the ability to directly start the student implementation using a Java Webstart launcher further helps the grading staff to assess the student's submission very efficiently.

Students also have the opportunity to get bonus points by pointing out bugs in our framework or documentation. We have found several small errors in our course material this way.

The necessary time spent by faculty members for evaluation talks is half an hour per student per semester. For 150 students, this is a total of 75 hours. Two graders could therefore hold the talks within one week per evaluation phase. In order to allow swift completion of the grading phases 15 members of the faculty help with holding the evaluation talks of our course. The effort for each individual grader is therefore relatively low with about five hours or twenty 15-minute evaluation talks per semester.

5.4. Submission system

The web-based online submission system is used for student registration, submission of assignment solutions, download of reference solutions and for grading.

After a submission package has been uploaded by a student, the archive file is unpacked and compiled. If there are any errors during this process, the student will be notified via email. The compiled student implementation is then executed by the submission system for rendering several reference scenes in a headless batch-mode.

In the first iteration of the course, students would get a comparison page that would show the generated images of their implementation side-by-side with images generated by the reference solution. We later removed this feature because we noticed that some students would debug their assignments in a *change–upload–compare* loop without really using the application GUI. This was undesirable because we want to encourage experimentation with the interactive viewer and because it creates an unnecessary load for our systems. We therefore disabled the comparison page for students. Students should instead download and compare the reference scenes themselves in the interactive

[4] https://lva.cg.tuwien.ac.at/ecg/wiki

application. The comparison pages are still visible for graders during evaluation talks.

Because we wanted to encourage experimentation in the second iteration of our course, the students are now required to create a simple scene during each assignment to showcase the features that were implemented. This file and a textual description of the scene is also packed and uploaded with the submission file and will be rendered using the student's own solution afterwards. The resulting image, along with the description of the scene, can be viewed in the submission system and a large overview page with images from all student scenes for a given assignment is created.

The submission system helps graders during the evaluation talks in several ways. The student's solution can be directly launched from the browser using Java Webstart. The task annotations of the reference source are also used to create bookmarks which can be used to directly jump to relevant source sections in the student's source code. In addition to that, the system is able to automatically create a color-coded comparison between one student's solutions to another. This allows to explicitly check for cheating, if necessary. Graders can also view the render comparison page to get a quick overview on the completeness of the solution.

6. Discussion

Our new introductory computer graphics course was held twice in the previous two years. We also taught the intermediate course each time in the following semester. In the intermediate course students write a computer game from scratch using C++ and OpenGL. We noticed that students who attended our new introductory course had less problems picking up shader programming with OpenGL and GLSL later. While we do not teach OpenGL in our introductory course, we take extra care to teach practical skills that are easily transferable from our didactic Java framework to real-world modern OpenGL programming. One example would be the interplay between the viewing pipeline and shaders, where the transformation matrices (model/view/projection) have to be explicitly assembled and passed to each shader instance in order to perform lighting calculations. While teaching the intermediate course we found that we did not have to hold tutorial sessions on basic shader programming anymore because students were already familiar with the concept and only had to learn how to apply it with the OpenGL API and GLSL. As a result we were able to spend more time on other topics of the intermediate course, such as advanced OpenGL aspects or game programming techniques. This also allowed us to make the use of a Core OpenGL 3.x profile mandatory in order to pass the intermediate course. Previously, we had also tolerated the use of the deprecated fixed-function OpenGL model.

There are commercial [27, 28] and didactic tools [16] that allow easy authoring and interaction with shaders which also could be used for teaching. WebGL [29], which is now available on all major web browsers, might also provide an interesting learning environment. However, those tools and frameworks use either OpenGL or DirectX for their implementation and hide the actual execution of the graphics pipeline from the student. We therefore believe that those might be very good for learning shader authoring or higher level OpenGL, but that they are not best suited for teaching the graphics pipeline. Since our course framework uses a didactic software rasterizer, students are able to both write shaders and understand how the graphics pipeline actually interacts with the shaders. For example, the students are able to set a breakpoint in the triangle rasterizer and observe how the fragment shader instance is called for each rasterized fragment. We believe that this ability is important and that it allows our students to better understand the graphics pipeline.

The framework has been used in two iterations of the course. In the first iteration 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 5, allowed us to quickly change the assignments and to adapt to this problem for the second iteration. For this we reduced the amount of student documentation and asked students to create test scenes to showcase the implemented features after each assignment. We have also changed later assignments to be more open. An example for this is the completely new last assignment, where students implement texture mapping and implement a custom effect shader.

The source code of the reference solution is only made available to students who are registered in the submission system. Despite this, there is still a possibility of students handing over their source code to students doing the course in the following semester. Possible solutions are regular variation of the course (framework code changes, replacement of algorithms, change of assignments) and use of automated source code plagiarism tools.

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 4 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.

We think that our framework covers a broad range of fundamental topics in computer graphics. There are, however, features that we have not implemented. Hierarchical transformations have been shown to be valuable to an introductory computer graphics course [9]. Adding this feature would require adapting the `scene` package to support object hierarchies. We chose not to support it so far because calculation of final model- and especially view-matrices would require a relatively complex recursive operation. We also do not 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.

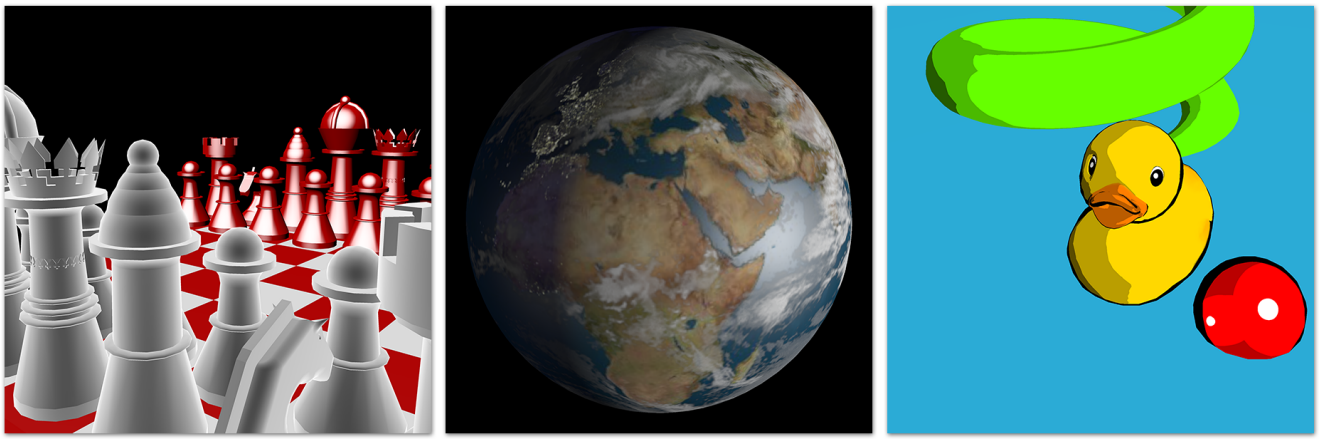

Figure 4: Entries submitted by students for the annual lab course competition. From left to right: A COLLADA-imported chess scene with custom shaders applied by Philipp Seeböck; Day-and-night shader implementation by Levin Pölser; Cel-shading by Sascha Wiplinger.

In our software design we have decided to generally favor code clarity and modularization over code-level optimizations. Rhodes and Yan implemented a similar course framework with EASEL, a didactic software-based rasterizer written in Java [24]. They recommend avoiding small allocations on the heap in favor of allocating larger blocks and reusing objects whenever possible. In many situations we trust Java's optimizing compiler to avoid potential performance impacts due to a higher-level class design. Optimizations that we build upon are for example escape analysis for moving local allocations to stack memory and inlining of virtual method invocations using just-in-time compilation.

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 [24]. This suggests that our choice of using easier-to-read and small classes does not result in a significant impact on performance in comparison with EASEL. Our framework also supports multi-threaded rendering by default which pushes frame rates for most models beyond 30 fps or higher, depending on the number of CPU cores that are available.

7. 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 increase 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.

8. Acknowledgments

We would like to thank our administrator 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. Another big thank you goes to all the tutors that helped us to carry out the course successfully.

[1] Fink H, Weber T, Wimmer M. Teaching a Modern Graphics Pipeline Using a Shader-based Software Renderer. Cagliari, Sardinia, Italy: Eurographics Association. ISBN -; 2012, p. 73–80.

[2] Ohlson MR. The role and position of graphics in computer science education. SIGCSE Bull 1986;18(1):232–7.

[3] Wolfe R. Bringing the introductory computer graphics course into the 21st century. Computers & Graphics 2000;24(1):151 –5.

[4] Hitchner LE, Sowizral HA. Adapting computer graphics curricula to changes in graphics. Computers & Graphics 2000;24(2):283 –8.

[5] Cunningham S. Powers of 10: the case for changing the first course in computer graphics. In: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education. SIGCSE '00; New York, NY, USA: ACM. ISBN 1-58113-213-1; 2000, p. 46–9.

[6] Peter I, Gumhold S. Teaching computer graphics with java 3d. Advances in Multimedia and Distance Education (Proceedings of ISIMADE'99) 1999;.

[7] Hitchner L, Cunningham S, Grissom S, Wolfe R. Computer graphics: the introductory course grows up. SIGCSE Bull 1999;31(1):341–2.

[8] Bouvier DJ. From pixels to scene graphs in introductory computer graphics courses. Computers & Graphics 2002;26(4):603 –8.

[9] Cunningham S, Bailey MJ. Lessons from scene graphs: using scene graphs to teach hierarchical modeling. Computers & Graphics 2001;25(4):703 –11.

[10] Cunningham S, Hansmann W, Laxer C, Shi J. The beginning computer graphics course in computer science. SIGGRAPH Comput Graph 2004;38(4):24–5.

[11] Sung K, Shirley P. A top-down approach to teaching introductory computer graphics. Computers & Graphics 2004;28(3):383 –91.

[12] Angel E, Cunningham S, Shirley P, Sung K. Teaching computer graphics without raster-level algorithms. ACM SIGCSE Bulletin 2006;38(1):266–7.

[13] Cunningham S. Computer graphics in context: an approach to a first course in computer graphics. In: ACM SIGGRAPH ASIA 2008 educators programme. SIGGRAPH Asia '08; New York, NY, USA: ACM. ISBN 978-1-60558-388-4; 2008,.

[14] Schweppe MK, Geigel J. Teaching Computer Graphics in the Context of Theatre. In: [30]; 2009, p. 67–72.

[15] Owen GS, Zhu Y, Chastine J, Payne BR. Teaching programmable shaders: lightweight versus heavyweight approach. In: ACM SIGGRAPH 2005 Educators program. SIGGRAPH '05; New York, NY, USA: ACM; 2005,.

[16] Bailey M. Teaching OpenGL shaders: Hands-on, interactive, and immediate feedback. Computers & Graphics 2007;31(3):524 –31.

[17] Talton JO, Fitzpatrick D. Teaching graphics with the openGL shading language. In: Proceedings of the 38th SIGCSE technical symposium on Computer science education. SIGCSE '07; New York, NY, USA: ACM. ISBN 1-59593-361-1; 2007, p. 259–63.

[18] Angel E, Shreiner D. Teaching a Shader-Based Introduction to Computer Graphics. Computer Graphics and Applications, IEEE 2011;31(2):9 –13.

[19] Roberts E, Engel G, Chang C, Cross J, Shackelford R, Sloan R, et al. 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;.

[20] Fuller U, Pears A, Amillo J, Avram C, Mannila L. A computing perspective on the Bologna process. SIGCSE Bull 2006;38(4):115–31.

[21] Bourdin JJ, Cunningham S, Fairn M, Hansmann W. Report of the CGE 06 Computer Graphics Education Workshop, Vienna, Austria, September 9, 2006 . Prague: Eurographics Association; 2007, p. 51–6.

[22] Mukundan R. Teaching computer graphics using Java. SIGCSE Bull 1999;31(4):66–9.

[23] Tori R, Bernardes Jr. JaL, Nakamura R. Teaching introductory computer graphics using java 3D, games and customized software: a Brazilian experience. In: ACM SIGGRAPH 2006 Educators program. SIGGRAPH '06; New York, NY, USA: ACM. ISBN 1-59593-364-6; 2006,.

[24] Rhodes PJ, Yan B. Easel: A Java Based Top-Down Approach to 3D Graphics Education. In: [30]; 2009, p. 29–36.

[25] COLLADA: Digital Asset and FX Exchange Schema. 2012. URL `http://www.khronos.org/collada/`.

[26] Google Warehouse. 2012. URL `http://sketchup.google.com/3dwarehouse/`.

[27] Tatarchuk N. RenderMonkey: an effective environment for shader prototyping and development. In: ACM SIGGRAPH 2004 Sketches. SIGGRAPH '04; New York, NY, USA: ACM. ISBN 1-58113-896-2; 2004, p. 91.

[28] NVIDIA FX Composer. 2012. URL `http://developer.nvidia.com/content/fx-composer`.

[29] WebGL: OpenGL ES for the Web. 2012. URL `http://www.khronos.org/webgl/`.

[30] Domik G, Scateni R, editors. EG 2009 - Education Papers. Munich, Germany: Eurographics Association; 2009. ISBN undefined.