Upgrading Tiny Graphics to WebGL 2.0 and Modern Rendering Standards

Adrien Hadj-Chaib
ahadjchaib@ucla.edu
Computer Science Department
University of California, Los Angeles

June 11, 2021

Abstract

We introduce a major update to the Tiny Graphics library, the centerpiece of UCLA's undergraduate curriculum in Computer Graphics. The objective of this overhaul is twofold. We first aimed at improving the performance of the library by upgrading it to WebGL 2.0, allowing reorganization of the rendering pipeline so as to minimize its CPU overhead. The second aim was to make Tiny Graphics adopt an industry standard rendering architecture that allows students to learn about best real-time rendering practices in a simple code-base. We then devised a comprehensive underwater scene rather easily following the new rendering architecture, showcasing the potential and efficiency of Tiny Graphics' new rendering organization.

1 Introduction

Tiny Graphics is an educational platform introduced by Ridge and Terzopoulos [2019] with the intent of simplifying the teaching of introductory computer graphics at the undergraduate level. Its strength lies in its small code-base and simplicity, providing students with a sandbox in which they can learn graphics by doing. Tiny Graphics has been used as the programming platform for assignments and term projects for UCLA's *CS174A - Introduction to Computer Graphics* course since 2015. Tiny Graphics underwent many improvements over the years, successfully enabling students to apply the theory learned in the classroom.

This work is in line with the continuous performance improvement of the library. Following from this, we upgraded the low-level API to WebGL 2.0 as well as used modern rendering techniques thereby reducing the number of expensive low-level calls. This overhaul also introduced an industry standard rendering architecture and incorporates container classes for the different scene elements. Having this single layer of abstraction allows students to approach graphics in terms of logic rather than being bogged down by low-level details. This also has the additional benefit of teaching best rendering practices, which are essential to follow when writing real-time graphics applications. It is important to note that students can still easily delve into the core implementation due to Tiny Graphics' preserved simplicity.

We then used Tiny Graphics to devise a comprehensive underwater scene rather easily following the new rendering architecture. Tiny Graphics' performance improvements showed a positive impact on the quality of work that students will be able to produce, allowing for scenes with more elements and more advanced effects such as real-time shadow mapping.

2 Motivations

2.1 Upgrade to WebGL 2.0

The original version of Tiny Graphics was powered by the WebGL 1.0 infrastructure which lacks modern rendering structures, such as Vertex Array Objects (VAO) and Uniform Buffer Objects (UBO). Upgrading Tiny Graphics to WebGL 2.0 allowed us to use Vertex Arrays to package and send geometry data to the GPU, as well as to use UBOs to send uniforms to the shader in an efficient manner. Such rendering architecture is the norm in modern graphics APIs. Incorporating it into Tiny Graphics exposes students to these concepts in a small environment that is easily understandable.

2.2 Performance Improvements

Aside from the performance gain from upgrading to WebGL 2.0, Tiny Graphics performed a lot of low-level API calls under the hood. Such low-level calls are expensive due to the extra validation that WebGL needs to impose in order to ensure web security. Accordingly, the CPU side overhead of running WebGL applications is known to be higher in comparison to native OpenGL applications. Graphics-heavy WebGL applications can become bottlenecked on the CPU side when interfacing with low-level functions [Emscripten, 2015]. We strove to reduce the number of low-level calls performed at each frame in order to achieve even greater performance.

2.3 Modern Entity System

The motivation for an Entity system is to provide an intuitive abstraction around which students can center their scene. Students previously had to think in terms of drawing a list of primitive Shapes by passing in the correct material each time, which is error prone. For instance, if a student wants to draw a football and a basketball in their scene, they previously had to call the sphere Shape's draw function twice, passing in a football or basketball material. Under the new paradigm, a student only needs to setup their football and basketball entities once and submit them to the Renderer each frame they wish to draw them. It is analogous to thinking "I wish to draw a football" rather than "I wish to draw a sphere that has the appearance of a football". Such abstracted and logical thinking will help students to create more complex scenes by feeling less tied down to low-level details from the Tiny Graphics API.

2.4 Modern Rendering Practices

Reducing Drawcalls Issuing drawcalls should be done with caution and understanding. Prior to the new Renderer and Entity System, students had to issue individual drawcalls for each geometry instance being rendered. Drawing consisted of activating the shader, setting the uniforms, and issuing the draw command once for every scene element. Each draw was an important source of CPU overhead. One core observation is that a scene often has a lot of repeated geometry with the same material, which can be rendered at once using instancing, thus greatly reducing rendering overhead. Instancing is a graphics card feature that became available upon our upgrade of Tiny Graphics to use WebGL 2.0. Instancing has the double benefit of achieving even better performance and also teaching students that minimizing the number of drawcalls using instancing or batching techniques is primordial to writing real-time rendering applications.

Entity System The aforementioned Entity System is an industry proven and efficient way to compose the elements of a scene. It is the basis of modern game engines such as Lumberyard from [Amazon Game Tech, 2021] or Unreal Engine from [Epic Games, 2019]. Exposing students to this data organization allows them to apply their prior knowledge of these tools to Tiny Graphics and beyond.

Material System We created a dedicated Material class to contain all shader data relative to a mesh's final color. This class was created to support the new Entity paradigm and shift student's perspective on Materials. Previously the Tiny Graphics API introduced materials as overriding uniforms in a drawcall. This viewpoint is not ideal because it teaches students to set a new material before every drawcall, even if the material does not change. The new Material class utilizes Uniform Buffer Objects (UBOs), another graphics card feature that became available upon our upgrade of Tiny Graphics to use WebGL 2.0. The Material class now equips students with a tangible material object that will dictate any geometry's final color, as in modern game engines. The Renderer can then group same-material Entities together to reduce the number of shader switches and uniform bindings between drawcalls.

Algorithm 1: Tiny Graphics Program Structure

```
Function Init:
    create Camera;
    create Lights;
    create Shaders;
    create Materials (Shaders);
    create Geometry;
    create Entities (Geometry, Materials);
    create Renderer;
End Function

Function RenderStep:
    update Camera;
    update Lights;
    update Materials;
    update Entities (TransformList);
    Renderer.Submit(Entities);
    Renderer.ShadowMapPass(Lights);
    Renderer.Flush(Lights);
End Function
```

Figure 1: Simplified workflow of a Tiny Graphics program, including the initialization and frame update phases.

Render Passes Modern rendering architectures are organized in passes, which can be seen as logical steps to construct a frame. For instance, it is common to start with a pass dedicated to updating the scene's shadow maps before rendering the scene's Entities, as the latter depends on the former. Tiny Graphics now encourages students to use the Renderer class, which is organized in passes that the student calls explicitly. This organization teaches students about frame construction while avoiding context switching so as to achieve better performance.

3 Methods

3.1 Program Architecture

Writing a program under the new Tiny Graphics framework can be summarized as in Figure 1. There is an initialization function wherein the user instantiates the data and logical objects used to create and render the scene. The second function is called per frame, where the user updates the data and logical objects. Once they have been updated, the user submits the Entities to be drawn to the Renderer. The user will then call the different Render Passes to composite the final frame using the Entities and Lights that have been submitted.

3.2 Entities

The basic building block of a scene in Tiny Graphics is a new object known as an Entity. An Entity's high level structure is depicted in Figure 2. An Entity is comprised of a Shape, the 3D Geometry that will be rendered. It also contains a Material, consisting of the shader program, color, textures, and lighting coefficients used to determine the final color of the Shape. Finally, an Entity also contains a list of Transforms, represented as an array of 4x4 Matrices. Each transform matrix represents one final position of one instance of the Shape, meaning that the Renderer will draw as many instances of the Shape as there are transform matrices, using only one instanced drawcall.

3.3 Renderer

The Renderer's role is to draw all the Entities that have been submitted to it for the current frame. When calling the `Submit(Entity)` function, the user is essentially appending the Entity to a list which

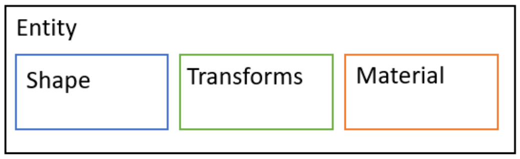

Figure 2: High-level structure of the Entity object.

Algorithm 2: `Renderer.flush()`

```
Function flush:
    foreach entity in Entities do
        if entity's transforms is a Matrix then
            Update the entity's shape "Matrix" Vertex Buffer Object;
            Draw one instance of entity's shape, using entity's material;
        end
        else if entity's transforms is an Array of Matrices then
            Update the entity's shape "Matrix" Vertex Buffer Object;
            InstancedDraw transforms.length of entity's shape, using entity's material;
        end
    end
End Function
```

Figure 3: Simplified workflow of the Renderer's `flush()`.

the Renderer will process once the user calls `flush()`. The `flush()` function can be summarized as in Figure 3. The Renderer also has a `shadowMapPass(lights)` function whose role is to render the already-submitted entities to the shadow maps of each of the shadow-casting lights. This results in a clear distinction between specialized render passes to update the shadow maps and to render the scene.

3.4 Shader Programs

Shader programs have been adapted to support VAOs as vertex shader input, allowing for a standardized input layout aligned with modern OpenGL. The second paradigm shift was to use UBOs rather than individual uniforms to pass in global shader inputs such as Camera, Material, and Lighting information. One advantage of using UBOs is that they make organizing input data into structures easy, leading to organized shader code from which students can learn easily. Furthermore, UBOs enable updating global shader data once per frame rather than once per shader program instance. This greatly reduces the number of low-level uniform binding calls occurring between drawcalls, resulting in an important performance gain. As a result, students learn that UBOs and uniforms serve different purposes and should be used for global and local scope, respectively.

3.5 3D Geometry

Shape Objects Upgrading Tiny Graphics to WebGL 2.0 gave us the ability to use VAOs to send input data to the vertex shader. This implied converting the Shape objects from using custom arrays to an interleaved VAO-based data layout, exposing students to using modern geometry data structures.

3D Meshes (.obj) Tiny Graphics already supported 3D assets with some very limited support for the .obj file format for specifying models. We brought some updates to this feature by also exporting the geometry data to a VAO-based data layout, essentially treating primitive shapes and imported meshes equally. This proves to be useful as an Entity can be constructed without differentiating the source of the vertex data. We also resolved an issue in the .obj loading logic leading to texture coordinates not being loaded correctly.

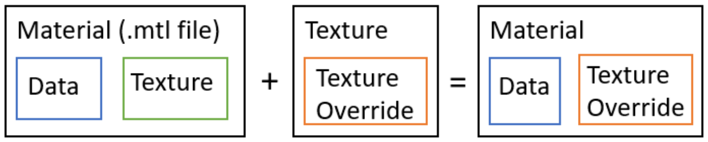

Figure 4: Illustration of .mtl file overriding.

3.6 Material System

Mesh Materials (.mtl) We devised a material loader analogous to the aforementioned .obj loader to support the importing of textured .obj models. Its aim is to load a .mtl file into a Material instance. An Entity can thus be constructed without differentiating the source of the material data. We further allow students to override the fields of a .mtl file as in Figure 4, enabling them to modify a loaded .mtl file without needing to alter the file.

Basic Material A basic Material contains a shader reference and data relative to a mesh's final color. This includes a base color, diffuse and specular coefficients, a smoothness term, as well as texture data (diffuse, normal, specular). All Material data is grouped in a UBO to the exception of texture samplers because they cannot be contained within a UBO. This architecture makes switching materials efficient by setting all the non-texture data at once through the UBO.

3.7 Lighting

Basic Light We created a Light class with which the user can create a directional or point light and specify its color as well as the usual Phong lighting coefficients. Having such an abstraction for lights serves the same purpose as materials in that it shifts the paradigm toward viewing a light as its own tangible object rather than some uniforms to override before each drawcall. It is also important to note that lighting information affects the scene globally, which is why all the lights in the scene are represented using a UB0 containing an array of lighting information where each instanced light has its own entry. This design allows updating all the lights once for all shaders, greatly reducing the number of required low-level calls.

Shadow-Casting Lights Inheriting from our Light class is a separate ShadowLight class supporting shadow-casting for both directional and point lights. ShadowLights hold an additional shadow map term consisting of a Frame Buffer Object (FBO) with a depth target, which is updated during the Renderer's ShadowMapPass function call. The shadow maps are internally used as an array of samplers outside of the UBO and are bound per shader activation. The reason for separating light classes is not to overwhelm students with the complexity of the shadow mapping implementation firsthand. A side benefit is that it gives students the freedom to re-implement shadow mapping in their term projects, while letting those who choose otherwise be able to have shadows in their projects.

3.8 Camera

We created a Camera class analogous to the Material and Light classes, in that students also think of the Camera as its own entity rather than some uniforms to override before every drawcall. The Camera class serves as a placeholder for the projection, view, and position values consumed by the shaders. We also used a UBO for the Camera as it affects the scene globally, and allows updating the Camera once for all shaders, requiring a minimal number of low-level calls.

4 Results

The updated Tiny Graphics library was used to create an underwater fish schooling scene as part of the Spring 2021 *CS275 - Artificial Life for Computer Graphics and Vision* course taught by Professor Demetri Terzopoulos. A live demo can be found at www.brandx.net/cs275/. The library enabled us

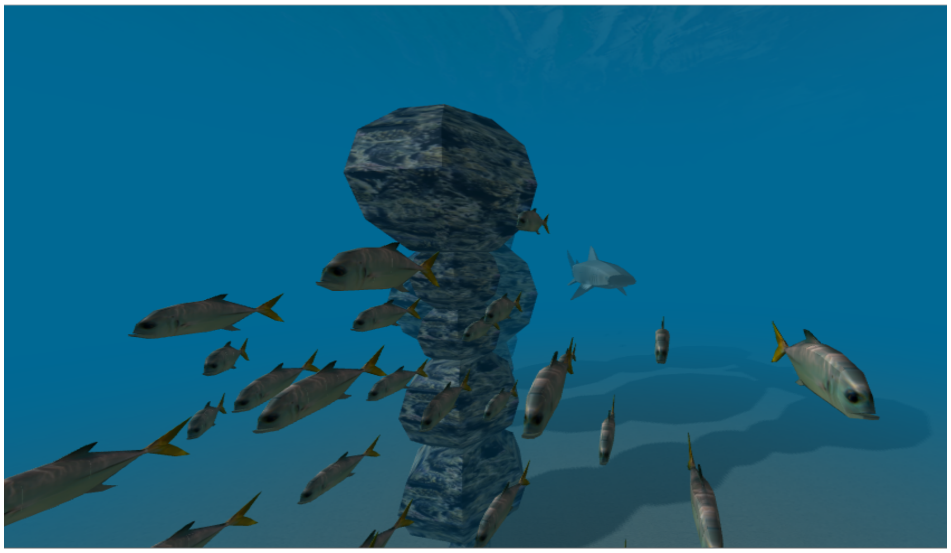

Figure 5: Underwater fish schooling scene.

to create a comprehensive scene while keeping the rendering code simple and organized following the workflow depicted in Figure 1.

Figure 5 showcases the entities comprising the scene (fish, shark, coral, underwater sky, sand floor), all of which are imported textured 3D models. There is also a unique shadow-casting directional light, whose shadow map is being updated every frame effortlessly. This allows the creation of interesting effects such as having shadows follow the movements of the fishes. The light source was also given an oscillatory movement to simulate the effects of surface waves on light direction. This makes the shadows of all objects oscillate, adding to the scene's realism. Finally, the blue fog effect was implemented in the fragment shaders, and is not a part of Tiny Graphics.

Reducing low-level calls and draw commands led to a significant performance improvement, allowing support for expensive features such as real-time shadow mapping. The use of instancing enables the rendering of all members of each category of entity in a single drawcall, including hundreds of fish with no noticeable slowdown. We found that performance noticeably decreases when approaching one thousand fishes as in Figure 6. This is due to the fact that collision detection is applied per fish, performing a distance test with all the remaining fishes. This creates a CPU bottleneck independent of Tiny Graphics' rendering capabilities, as collision scales quadratically with the number of fish whereas rendering remains constant. To confirm this result, we disabled the fish collision and steering logic, allowing us to support more fish smoothly as shown in Figure 7.

5 Conclusions and Future Work

We have successfully upgraded the Tiny Graphics library to use the newer WebGL 2.0 API. We organized the rendering pipeline to minimize the number of low-level calls, greatly improving performance. The effort serves two purposes—making students achieve more with the tool and exposing them to rendering optimizations. This allows students to build a valuable cost-model of rendering operations, such as learning that issuing drawcalls or binding uniforms are costly and should be done carefully.

A secondary objective was to overhaul the library to the use of a modern rendering architecture centered around an Entity system and a Renderer class. This new organization better reflects modern rendering practices, giving students the opportunity to learn these basic principles in a smaller and self-contained environment. Introducing logical objects such as a Camera and Material will also help students understand the bigger picture of these systems as opposed to first seeing them as low-level details. Once students understand the basic principles, delving into low-level implementation will be seamless thanks to the Tiny Graphics philosophy of remaining a small code-base.

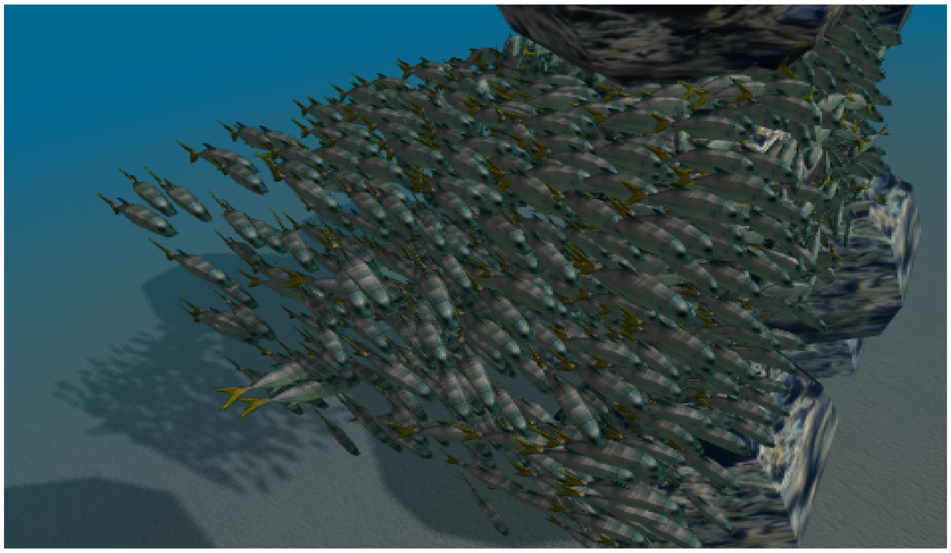

Figure 6: Enabling external collision logic allows the smooth rendering of 1,000 fishes in a single drawcall.

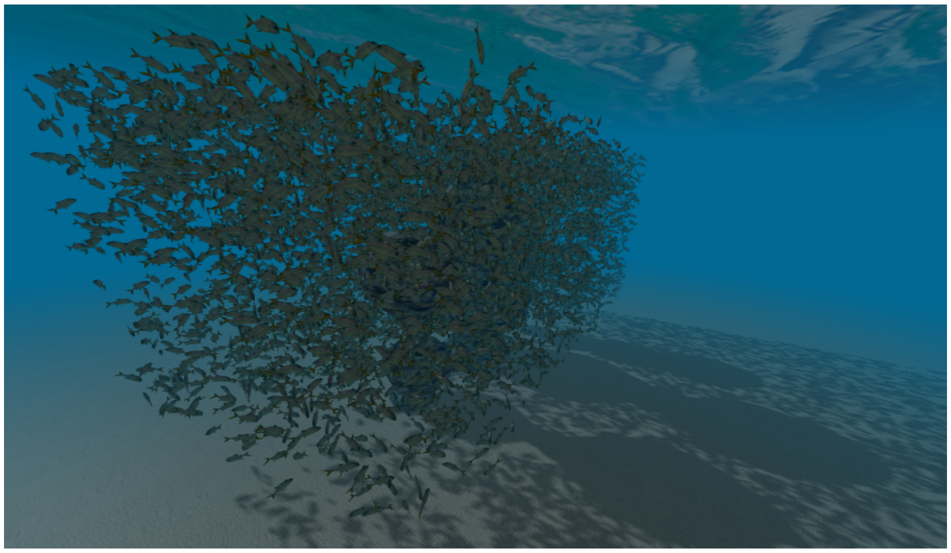

Figure 7: Disabling external collision logic allows the smooth rendering of 8,000 fishes in a single drawcall.

5.1 Future Work

We plan to add other features to make Tiny Graphics a more complete and efficient library capable of handling student's project ideas effortlessly.

Renderer.Draw We plan on relocating the `Shape.draw()` capability to a dedicated Renderer function that can draw any Shape. Having the `draw()` function as part of the Shape object is one of the reasons that led students to infer that rendering one shape instance equated one drawcall.

Entity Batching In order to further minimize the number of low-level rendering calls, we can render all Entities sharing a same Material before using a different one. Doing so minimizes the number of shader activation and Material bindings. If two Entities use the same Material, we can further batch their transforms together if they use the same Shape geometry, further reducing the number of drawcalls.

Model Loading We aim to improve .obj loading to correctly load sub-meshes as we currently only support models comprised of a single mesh. Material loading is already able to load all the material specifications in a single .mtl file. We will therefore need to link sub-meshes to the correct material, and update the Entity to support multiple Shape and Material objects accordingly.

Transform Feedback Transform Feedback is used for implementing particle systems, an important feature of any modern rendering system that students can also use in their projects. Additionally, Transform Feedback teaches students about double-buffering optimizations, as well as the ability to offload transform generation to the vertex shader.

Acknowledgement

I would like to thank Jim Pickrell, Tomoyori Iwao, and Wenqi Zou for their integral help in devising the underwater scene showcased in this project, as well as for making it publicly available online. I would also like to thank Dr. Garett Ridge for agreeing to the idea of overhauling Tiny Graphics, for his availability through many all-nighters, as well as for his essential help throughout this project. x)

References

Garett D Ridge and Demetri Terzopoulos. An online collaborative ecosystem for educational computer graphics. In *Proceedings of the 24th International Conference on 3D Web Technology (Web3D)*, pages 1–10, Los Angeles, CA, July 2019.

Emscripten. Optimizing WebGL. https://emscripten.org/docs/optimizing/Optimizing-WebGL, 2015. Accessed: 2021-06-10.

Amazon Game Tech. Amazon Lumberyard, May 2021. URL https://aws.amazon.com/lumberyard/.

Epic Games. Unreal Engine, April 2019. URL https://www.unrealengine.com.