Rayground: An Online Educational Tool for Ray Tracing

N. Vitsas[1], A. Gkaravelis[1], A. A. Vasilakis[1], K. Vardis[1], G. Papaioannou[1]

[1]Department of Informatics, Athens University of Economics and Business, Greece

Abstract

In this paper, we present Rayground; an online, interactive education tool for richer in-class teaching and gradual self-study, which provides a convenient introduction into practical ray tracing through a standard shader-based programming interface. Setting up a basic ray tracing framework via modern graphics APIs, such as DirectX 12 and Vulkan, results in complex and verbose code that can be intimidating even for very competent students. On the other hand, Rayground aims to demystify ray tracing fundamentals, by providing a well-defined WebGL-based programmable graphics pipeline of configurable distinct ray tracing stages coupled with a simple scene description format. An extensive discussion is further offered describing how both undergraduate and postgraduate computer graphics theoretical lectures and laboratory sessions can be enhanced by our work, to achieve a broad understanding of the underlying concepts. Rayground is open, cross-platform, and available to everyone.

CCS Concepts

• ***Social and professional topics*** → ***Computer science education;*** • ***Computing methodologies*** → ***Ray tracing;*** • ***Software and its engineering*** → *Software prototyping;*

1. Introduction

While ray tracing is one of the most common teaching subjects for both introductory and advanced computer graphics courses from around the world [BWF17], specialised educational tools to improve the learning curve on this topic in a way that attracts and engages students are still missing. Typical undergraduate graphics syllabi, build the structure of the courses around the rasterization pipeline and in the best case, devote a limited number of lectures to explain the basic paradigm of ray tracing, usually as the last part of the course. However, in recent years, ray tracing has gained significant momentum as a compelling alternative for achieving both the desired level of photorealism in production and interactive rendering and as the means to study shading algorithms [KVBB*19]. The advent of mass-produced, consumer grade hardware with ray tracing acceleration capabilities has significantly boosted the interest of the graphics community and has led to the introduction of related methods to interactive applications, thus demonstrating its wide applicability to students. Unfortunately, this turn of interest to ray-tracing-based techniques is not sufficiently backed by proper educational tools to assist students in becoming familiar with the basic concepts and help them become practically engaged in building their own projects. Moreover, modern low-level graphics APIs, either dedicated to ray tracing like NVIDIA OptiX [PBD*10] or supporting it, such as Microsoft DirectX 12 [WM19] and Khronos Group Vulkan [Sub18], pose high entry barriers to students and require a very daunting and long learning process, riddled with many distracting technicalities.

The World Wide Web is undoubtedly the medium with the biggest global outreach. The sandboxed environment in modern Web browsers offers one of the best platforms for the deployment of educational tools. Web-based applications like *Jupyter Notebook* [KRKP*16] have revolutionised interactive data science and scientific computing across many programming languages by giving the ability to edit, execute and preview code from the browser. Following this trend, computer graphics and visualisation have greatly benefited from similar solutions [MKRE16]. Currently, hardware accelerated computer graphics on the Web are only possible through the WebGL W3C standards. Aside from the fact that these are fairly low level APIs, there is no functionality exposed that accommodates ray tracing solutions. Web-based API solutions like the *BabylonJS* [CRLR14] and *Three.js* [Cab10] frameworks are very well designed libraries that make graphics programming easier by taking care of low-level details. However, these frameworks focus on game/application development making them too abstract to facilitate learning about the underlying principles of computer graphics. Last but not least, *ShaderToy* [JQ14] is a well-known and highly successful online tool for creating and sharing fragment shaders through WebGL, used both for learning and teaching 3D computer graphics. However, these solutions have been explicitly designed for rasterization-based development leaving no room for ray tracing experimental prototyping.

We introduce *Rayground*, an interactive education tool for richer in-class teaching and gradual self-study that provides a convenient introduction into ray tracing programming. Rayground abstracts the functionality of the underlying ray tracing stages to an extent

that still preserves the main concepts taught in a computer graphics course as well as eases the development of advanced visual effects in student projects. This work aims to demystify ray tracing fundamentals while relying on the established GLSL shading language for code development and the underlying WebGL pipeline for its hidden execution model. It is intended for students who are already familiar with the basics of computer graphics theory (geometry representation, transformations, basic shading, etc.) and shader-based programming. It has not been designed and developed to be a complete replacement of teaching computer graphics with modern shader-based programming, e.g. OpenGL/WebGL [Ang17], but rather as a complementary educational resource that harmoniously enriches the teaching environment. Rayground does not rely on any browser plugins and thus runs on any platform that has a modern standards compliant browser.

2. Related Work

Teaching computer graphics can be challenging due their dependence on a wide range of theoretical knowledge and practical skills, such as mathematics, physics and programming. As such, various approaches to teaching introductory computer graphics have been documented in the literature over the years in order to transform teaching from a passive knowledge transmission to a more active and engaging process. For example, in-class interactive illustrations [SÅAM17] and rapid exercises [WD15] could result in a more effective understanding of the course material and the underlying mathematics. While the theoretical goals of the main course in computer graphics remain largely unchanged, graphics software technology has significantly evolved to support the tremendous advances in hardware [Ang17].

In today's typical computer graphics syllabi, most subjects are presented through the rasterization pipeline [RME14], which is both approximate by nature and limited due to its strictly isolated local computations, making topics like visibility determination for light sources and environment sampling more complex to introduce. Ideally, shifting the practical example implementation to ray tracing would facilitate better presentation of topics such as parametric, procedural and analytic geometry visualisation, including constructive solid geometry operations and volume graphics. Although ray tracing is one of the most common teaching subjects for both introductory and advanced computer graphics courses from around the world [BWF17], *Ray tracing in one weekend* book series [Shi19] is the only valuable resource available to help novice students start coding the very basics. On the other hand, a myriad of education solutions have been developed by the academia for teaching the traditional rasterization pipeline paradigm, using shader-based programming.

Learning computer graphics techniques through plugin development has multiple advantages: it allows for very focused, self-contained, independent exercises, it enforces modularity and facilitates code reusability. Fink et al. [FWW12] presented a syllabus for an introductory computer graphics course using Java that emphasises the use of programmable shaders, while teaching rasterisation-related algorithms. Shaders are implemented as classes and interact with the software rasteriser pipeline through polymorphism, in order to help novice students adopt the modern approach of shader-based programming patterns. Reina et al. [RME14] designed a GPU-accelerated educational framework that enables students to write code targeting modern OpenGL, exclusively. Each assignment is developed as a plug-in for this framework. In a similar fashion, Andujar et al. developed *GLsocket* [ACFV18], a flexible plugin-based C++ framework that offers four types of modules depending on their main purpose and the subset of methods they override including Effect, Draw, Render and Action.

A project-based learning direction can provide a constructive and motivational learning platform for computer graphics [Rom13]. Papagiannakis et al. [PPGT14] introduced *glGA*, a simple, thin-layer, open-source framework that curbs the computer graphics complexity by easily allowing students to grasp the basics through four simple examples and six sample assignments. Driven by this trend, the *FUSEE* [MG14] and *bRenderer* [BSP17] educational rendering frameworks hide non-graphics-related functionality to an extent that still allows students to easily grasp the concepts and techniques being taught.

Several computer graphics courses have moved to a Web-based educational programming environment in order to keep students with very different backgrounds engaged [FP13], by shifting the focus from low-level OpenGL API to object-oriented 3D graphics frameworks [AB15, RT19]. From a teaching perspective, WebGL offers a number of attractive features [Ang17] including among others, multi-platform support, easy integration with other Web APIs and strong student familiarity with Web technology.

Following ShaderToy's design [JQ14], Toisoul et al. [TRK17] introduced *ShaderLabFramework*, an integrated desktop development environment for a fast, programmable shading pipeline on a comprehensive lab exercise for undergraduate students. While two GPU ray tracing tasks were included in their course, they can only handle simple procedural objects that are easy to describe inside a fragment shader.

3. The Online Platform

Rayground is an online integrated development environment (IDE) for interactive demonstration and/or prototyping of ray tracing algorithms. Rayground, hosted at https://rayground.com, is free for everyone, on any platform that has a WebGL2-compliant browser (no special plugins are required). In general, the user has the ability to create any number of new projects from scratch or copy an existing one from a variety of ray tracing projects made available from other users. Since Rayground IDE is web-based and online, users can work on it from anywhere, anytime.

The graphical user interface of Rayground is designed to have two discrete parts, the *preview window* and the *shader editor* (Fig. 1), similar to the layout of ShaderToy [JQ14], which many shader developers are already familiar with. Visual feedback is interactively provided in the WebGL rendering context of the preview canvas, while the user performs live source code modifications.

Rayground follows a programmable GPU-accelerated ray-tracing pipeline (Sec. 3.1) in order to give developers direct and flexible control of five ray tracing stages through a simple, high-level shader-based programming model (Sec. 3.2). Thus, the shader

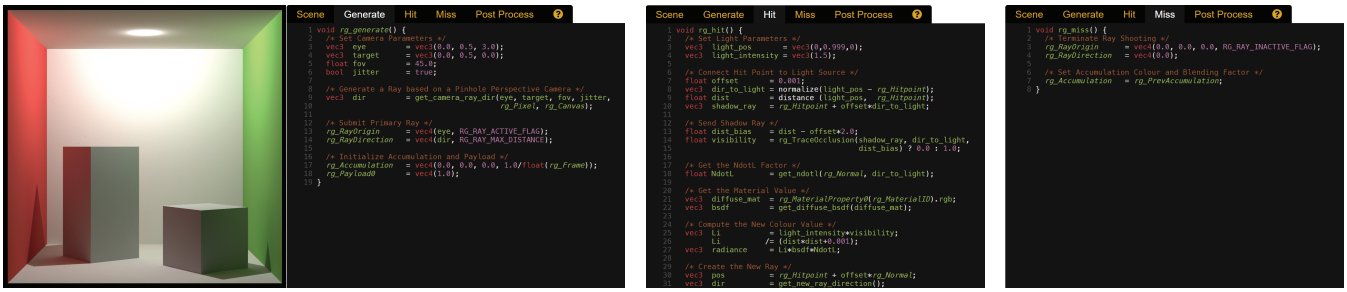

Figure 1: *Left: The Rayground interface, with the preview window and the shader editor, showing the Generate stage. Middle and Right: Hit and Miss event shaders for the same project, where the Cornell Box scene is rendered using a simple path tracer described in Sec. 3.3.*

editor consists of five tabs, corresponding to five customisable shader stages. A detailed documentation of the Rayground's programming interface is available on-site, while a summary of the basic functions and variables exposed by the Rayground API are accessible via the editor (see '?' tab in Fig. 1). The documentation, coupled with the many demo and tutorial projects, eases the way of newcomers and promotes self-study.

3.1. Ray Tracing Pipeline

At the core of Rayground there is a traditional ray tracing image synthesis pipeline, with several programmable stages via event handling shaders. It was designed with the aim to help users gradually understand how a ray tracer works, without getting distracted by the particular implementation of the framework or platform-specific characteristics. Since ray tracing is now tightly integrated into modern real-time rendering APIs [PBD*10,Sub18,WM19], we follow a similar programming model. Rayground's pipeline has five distinct configurable stages, namely *Scene*, *Generate*, *Hit*, *Miss* and *Post Process*, which are explained below, focusing on function rather than implementation.

The geometric objects of the scene are initially specified in the input *Scene* declaration stage. These objects are used to build ray intersection acceleration data structures, which in our case, are not programmable. Primary rays, which, in the simplest case correspond to a virtual camera, are generated and submitted for intersection in the *Generate* stage. Depending on the intersection results, execution switches to the closest *Hit* or *Miss* stage. Both events can generate a new ray which, in turn, may be intersected with the scene to trigger new events.

Users are provided with several built-in and user-controlled properties that ease the data transmission between events (Sec. 3.2). For each iteration of the pipeline, or *frame*, a pixel colour is computed and blended with the previous values stored in an *Accumulation Buffer*. All code segments execute in parallel for each pixel of the *Canvas*, i.e. the preview window and, in every frame, the executed code directly corresponds to the one iteration event, i.e. one ray path. The intermediate image is finally filtered through a *Post Process* stage, a common step prior to image presentation, handy for tone mapping and filtering operations. A graphical illustration of the pipeline is shown in Figure 2.

3.2. Application Programming Interface (API)

Thr Rayground API is implemented using the WebGL2 standard, supporting shader programming via GLSL, thus providing a convenient and familiar code development interface. The user is encouraged to use built-in GLSL functions (e.g. *dot*, *cross*) and types (e.g. *vec4*, *mat4*). However, any use of the standard input and output variables of the GLSL programmable pipeline stages (e.g. *gl_FragCoord*) as well as samplers (e.g. *sampler2D*) may result in undefined behaviour and should be avoided. While certain functionality is common to all stages, there are also stage-specific input and output variables, which are described below in more detail. The basic functionality of Rayground API is listed in Table 1.

A ray is defined with an origin point and a direction by setting the *rg_RayOrigin* and *rg_RayDirection* variables respectively. The ray is marked as active when *RG_ACTIVE_RAY_FLAG* is set at the fourth coordinate of the *rg_RayOrigin* vector. Note that the recursive ray shooting of each pixel can be terminated either by exceeding the maximum ray path (defined at the previous stage) or by submitting an inactive ray in any of *Generate*, *Hit* or *Miss* stages. The term ray depth, accessible by *rg_Depth*, is used to indicate the number of rays that have been shot recursively along a ray path. A maximum intersection distance is also required and is set using the fourth component of *rg_RayDirection*. The constant value *RG_RAY_MAX_DISTANCE* can be used instead, in order to use an unbounded ray. Subsequent stages depend on those output values and the user must be careful to initialise them for all pixels and all paths. Neglecting to do so, can result in undefined behaviour. The user can optionally add a *payload* to the ray via the *rg_Payload0* variable. This is a data structure that is used for relaying data between different stages. Last, *rg_Accumulation* carries the final colour of the ray, where the alpha channel holds the blending factor. Upon ray termination or after the maximum ray recursion is reached, the final ray colour is combined with the results from the previous frames in *rg_AccumulatedImage*, using additive blending. By manipulating the blending factor on the alpha channel of the ray colour, different results can be achieved such as simple value replacement or averaging of the values over all frames.

Scene stage. Specifying the geometry of a scene is one of the most basic tasks related to 3D visualisation. Rayground uses a simple custom JSON format, which is easy to manage and extend. A valid scene description contains *settings* and *objects* entities. The *set-*

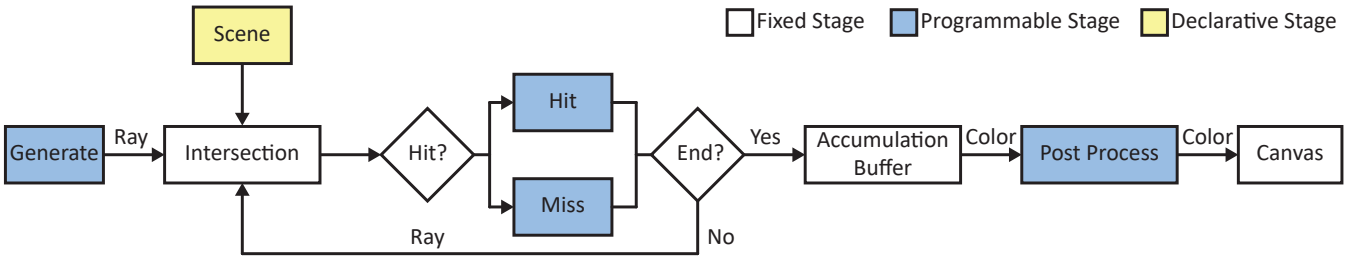

Figure 2: *High level overview of the Rayground pipeline. Shaded polygons correspond to programmable stages. Rays are submitted as waves and intersection results are provided to the next stages through appropriate API calls.*

tings object is used to configure pipeline options. In particular, the mandatory *depth* property sets the ray-tracing maximum recursion level. The *objects* entity specifies the scene's geometry. Most of the properties for each object are straightforward. The *type* field indicates the geometric type of the object. Several fundamental geometric types are supported including *quad*, *cube*, *sphere*, and *triangle* meshes. The *translate*, *rotate* and *scale* properties are used to position and orient an object in 3D space using a translation-rotation-scale transformation matrix. If a *model* matrix is present, it is used instead. In the special case of (analytical) spheres, *translate* is used to position the sphere in world coordinates and *radius* to change its scaling. Material properties for each object are supplied using *material_property* entries per object. Rayground supports up to eight generic material properties of *float[4]* storage type. These values can be used for various purposes from defining BRDF values to distinguishing between different material types. Their values are available to the user, when a ray-object intersection occurs.

Generate stage. This stage is responsible for the generation of primary rays, typically from a virtual camera and is the entry point for every project. Rays are spawned in parallel for each pixel of the image using a custom ray generation function called *rg_generate()*.

Hit stage. This stage is triggered when a ray hits a geometric object. Here, the closest hit point along the cast ray is provided in *rg_Hitpoint* and several of the geometric properties at the intersection point are accessible through special API calls (e.g. *rg_Normal*). Payload value of the ray is also available and the user can shoot a new ray in the scene using a custom ray-object function called *rg_hit()*. Local shading commonly takes place in this stage using the material properties of the intersected object, *rg_MaterialPropertyI(rg_MaterialID)*, $I = [0,7]$, as defined in the *Scene* stage. Note that a new ray can be spawned similar to the *Generate* stage, while access to the previous ray values are also provided using the *rg_Prev...* prefix. Last but not least, visibility queries between any two points can be performed using the *rg_TraceOcclusion()* function.

Miss stage. A ray that does not hit any geometric object triggers a custom ray miss function named *ray_miss()*. Although it does not have access to any information regarding geometric properties, it is allowed to generate new rays. This is useful for simulating light propagation in participating media or for interacting with other forms of procedurally modelled geometry. Most of the time, this stage is used for calculating the background colour.

Post Process stage. Modifying the image after rendering is as important in ray tracing as in every other image synthesis pipeline. Users may apply a number of full-screen filters and effects to the accumulated off-screen buffer, named *rg_AccumulatedImage*, before the image appears on screen. The *rg_PixelColor* variable, when defined in the *rg_post_process()* function, updates the final colour that is presented in the canvas.

3.3. Basic Example

We provide here a basic example, in order to present the pipeline and demonstrate Rayground's API mechanisms. It implements a simple unidirectional path tracer for diffuse surfaces in only a few lines of code. For clarity, the code of certain functions, commonly

Table 1: *Summary of basic Rayground API functionality.*

Type	Name	Description
All stages		
vec2	*rg_Canvas*	canvas resolution in pixels
vec2	*rg_Pixel*	pixel coordinates
int	*rg_Frame*	frame counter
Generate/Hit/Miss stages		
vec4	*rg_[Prev]Accumulation*	ray (prev) accumulation color
vec4	*rg_[Prev]Payload0*	ray (prev) payload
vec4	*rg_[Prev]RayDirection*	ray (prev) direction
vec4	*rg_[Prev]RayOrigin*	ray (prev) origin
int	*rg_Depth*	ray depth
bool	*rg_TraceOcclusion(...)*	ray occlusion query
Generate stage		
void	*rg_generate()*	entry point signature
Hit stage		
void	*rg_hit()*	entry point signature
vec3	*rg_Hitpoint*	hit in world space coordinates
vec3	*rg_Normal*	primitive's geometric normal
int	*rg_MaterialID*	primitive's material ID
vec4	*rg_MaterialPropertyI(...)*	material properties, $I = [0,7]$
Miss stage		
void	*rg_miss()*	entry point signature
Post Process stage		
void	*rg_post_process()*	entry point signature
vec4	*rg_PixelColor*	final pixel colour
rg_Image2D	*rg_AccumulatedImage*	2D accumulation image handle
vec4	*rg_ImageFetch2D(...)*	retrieve texels from 2D image

encountered in a typical path tracer, has been omitted (highlighted in light blue in the code listings below). Figure 1 illustrates how the *Cornell Box* scene is illuminated using this code after 500 samples per pixel. For more details, the interested reader can find the complete working example here.

In the *Scene* stage, the *Cornell Box* scene is trivially represented using five quads and two cubes for the diffuse geometry. The material properties are configured accordingly to set the colour for diffuse surfaces. The maximum iteration depth of the scene is set to five since additional bounces will add negligible radiance contribution. Due to paper length limits, we omit the corresponding JSON code. After the scene description, the *Generate* stage is used to implement a simple pinhole perspective camera. The *rg_Pixel* and *rg_Canvas* attributes are used to calculate a ray going through the centre of each pixel for a perspective camera with a 45° field of view. The generated rays are then submitted for intersection after preparing their *rg_RayOrigin* and *rg_RayDirection* variables. Additionally, the *rg_Accumulation* and *rg_Payload0* values are also initialised for each generated ray at this step. In a following fixed stage, the intersection of the primary rays with the scene entities is performed by the core ray tracing engine and the corresponding events for each event (hit or miss) are subsequently triggered.

```
void rg_generate() {
  /* Set Camera Parameters */
  vec3  eye          = vec3(0.0, 0.5, 3.0);
  vec3  target       = vec3(0.0, 0.5, 0.0);
  float fov          = 45.0;
  bool  jitter       = true;

  /* Generate a Ray based on a Pinhole Perspective Camera */
  vec3  dir          = get_camera_ray_dir(eye, target, fov, jitter,
                                          rg_Pixel, rg_Canvas);

  /* Submit Primary Ray */
  rg_RayOrigin       = vec4(eye, RG_RAY_ACTIVE_FLAG);
  rg_RayDirection    = vec4(dir, RG_RAY_MAX_DISTANCE);

  /* Initialize Accumulation and Payload */
  rg_Accumulation    = vec4(0.0, 0.0, 0.0, 1.0/float(rg_Frame));
  rg_Payload0        = vec4(1.0);
}
```

The *Hit* stage, called upon valid ray-object intersections, demonstrates the radiance accumulation process for the diffuse Monte Carlo path tracer using *next event estimation*; the direct illumination of each ray path is estimated by testing the intersected point for visibility with a punctual light source, using a shadow ray, and accounting for its contribution to the incident radiance. The important steps at this stage are the computation of the incoming illumination at the intersected point, the generation of a new ray and the storage of the light throughput at the current ray path. These computations can be trivially accomplished using the provided Rayground variables. For example, information about the intersected point can be obtained using *rg_Hitpoint*, *rg_Normal* and *rg_MaterialProperty*. To check for visibility, the *rg_TraceOcclusion()* casts a shadow ray between the intersection point and the light source position. Two final operations are necessary to complete the shader. First, new rays are submitted for intersection by writing the appropriate values to *rg_RayOrigin* and *rg_RayDirection* and making sure the *RG_ACTIVE_FLAG* is set in the fourth coordinate of *rg_RayOrigin*. Second, the *rg_Accumulation* and *rg_Payload0* variables are updated to store the gathered radiance and the accumulated light throughput, recursively.

```
void rg_hit() {
  /* Set Light Parameters */
  vec3  light_pos       = vec3(0,0.999,0);
  vec3  light_intensity = vec3(1.5);

  /* Connect Hit Point to Light Source */
  float offset       = 0.001;
  vec3  dir_to_light = normalize(light_pos - rg_Hitpoint);
  float dist         = distance (light_pos,  rg_Hitpoint);
  vec3  shadow_ray   = rg_Hitpoint + offset*dir_to_light;

  /* Send Shadow Ray */
  float dist_bias    = dist - offset*2.0;
  float visibility   = rg_TraceOcclusion(shadow_ray, dir_to_light,
                                         dist_bias) ? 0.0 : 1.0;

  /* Get the NdotL Factor */
  float NdotL        = get_ndotl(rg_Normal, dir_to_light);

  /* Get the Material Value */
  vec3  diffuse_mat  = rg_MaterialProperty0(rg_MaterialID).rgb;
  vec3  bsdf         = get_diffuse_bsdf(diffuse_mat);

  /* Compute the New Colour Value */
  vec3  Li           = light_intensity*visibility;
        Li          /= (dist*dist+0.001);
  vec3  radiance     = Li*bsdf*NdotL;

  /* Create the New Ray */
  vec3  pos          = rg_Hitpoint + offset*rg_Normal;
  vec3  dir          = get_new_ray_direction();
  rg_RayOrigin       = vec4(pos, RG_RAY_ACTIVE_FLAG);
  rg_RayDirection    = vec4(dir, RG_RAY_MAX_DISTANCE);

  /* Update Colour and Throughput Values */
  rg_Accumulation    = rg_PrevAccumulation;
  rg_Accumulation.rgb += rg_PrevPayload0.rgb*radiance.rgb;

  float  pdf         = get_pdf(rg_Normal, dir);
  float  NdotI       = max(0.0, dot(rg_Normal, dir));
  vec3   throughput  = bsdf*NdotI/pdf;
  rg_Payload0        = rg_PrevPayload0;
  rg_Payload0.rgb   *= throughput.rgb;
}
```

The *Miss* stage is called when an intersection with the scene is not found. In this example, the shader simply terminates the ray shooting and sets the incoming colour value to the *rg_Accumulation* buffer.

```
void rg_miss() {
  /* Terminate Ray Shooting */
  rg_RayOrigin       = vec4(0.0, 0.0, 0.0, RG_RAY_INACTIVE_FLAG);
  rg_RayDirection    = vec4(0.0);

  /* Set Accumulation Colour and Blending Factor */
  rg_Accumulation    = rg_PrevAccumulation;
}
```

Finally, the *Post Process* stage applies a basic *gamma correction* filter to each cell of the accumulated buffer and assigns the resulting values to the corresponding output canvas pixel through *rg_PixelColor*.

```
void rg_post_process() {
  /* Get Accumulation Buffer */
  vec4 image     = rg_ImageFetch2D(rg_AccumulatedImage,
                                   ivec2(rg_Pixel));
  /* Perform Gamma Correction */
  rg_PixelColor = gamma_correction(image, 2.2));
}
```

4. Teaching with Rayground

For us, the need to implement a platform that would enable the development of ray tracing-based algorithms arose primarily in the context of the graduate courses in advanced computer graphics,

where the majority of the lectures addressed photorealistic rendering. Developing a decent code base for the student projects, with an API that could be quickly picked up was really a problem; students typically come from slightly different undergraduate studies and one could not expect the same level of competence in a programming language like C++, let alone delve into the details of the currently available ray tracing APIs. We have tried to abstract ray traversal details by introducing a high-level hybrid CPU/GPU API in C++ and used it for a single semester to let students develop a photon mapping method. It turned out that, although most students had no difficulty mapping the required concepts to the event handles offered by the API, they invariably experienced difficulties with the general application development and debugging.

Rayground solved this problem very effectively, by eliminating the need for the students to familiarise themselves with a) the details of application development for their projects and b) another elaborate, platform-specific ray tracing API. Furthermore, it solved common issues of development inconsistencies between lab equipment and student development machines and software. It ensured that a steady, commonly available and highly accessible development platform was provided to the students. The use of a simple, event-driven coding paradigm with a GLSL API, levelled the ground for most students, who typically become familiar in shader development either through OpenGL- or, more recently, Vulkan-based lab coursework.

4.1. Undergraduate Syllabus Restructuring

Ray tracing is an important topic in computer graphics but, due to its complexity and lack of tools to demonstrate its functionality, it is usually discussed briefly during the last couple of lectures in an undergraduate computer graphics course. As ray tracing-based production rendering algorithms and real-time illumination techniques gradually establish themselves as a viable and more accurate replacement for approximate methods, in our lectures we have elevated the importance of ray tracing. Specifically, we have introduced the notion of pipeline abstraction for most of the topics related to rendering and have dedicated more class hours to teaching the basics of ray tracing, simplified data acceleration structures and a comparative study of ray tracing and rasterisation.

We divided the rendering pipeline into four generic stages: a) geometry setup, b) sampling/token generation, c) shading and d) compositing, with a specific note on the re-entrant nature of any of these stages. This allowed us to independently map rasterisation or ray tracing and clearly separate common topics from the specific pipeline, such as material properties, local shading, geometry representation, texturing, image-domain sampling and antialiasing. Rayground assisted in this abstraction process by offering an accessible alternative platform for in-class algorithms and principles demonstration (see below), while providing a simpler solution for the practical and accurate experimentation with certain concepts, such as visibility testing and material properties. Conversely, topics like texture anti-aliasing filtering are best explained through the rasterisation pipeline (here, the mip-mapping mechanism). Having a choice of paradigm, helps the students focus on the problem and gain access to easy to grasp solutions.

4.2. In-class Lectures

Rayground can be used to interactively demonstrate the concepts of ray tracing during lectures, to enrich and complement the theory presented in static slides. Allowing students to actively experiment using their laptops in class further transformed the learning process into a more active and engaging one. As reported by the students themselves, the learning experience was highly enhanced since most of them successfully correlated the presented concepts with their practical implementation and results. In an immersive fashion, students clarified with ease the recursive nature of ray tracing and how rays are generated and scattered though the virtual environment. Furthermore, complex mathematical models for physically-based lighting were effectively explained by previewing the effect when dynamically interacting with a specific parameter (e.g. roughness, index of refraction) inside a programmable stage.

4.3. Lab Coursework

As is very frequently the case, our computer graphics lectures are supported by a series of practical exercises in a lab, where students are able to apply the theory and experiment using simple frameworks. The contents of the lab sessions are aligned with the corresponding week's lectures. Sessions start with an introduction to the topics that will be discussed, followed by an exercise that is performed during the rest of the course. The lab assistant is present during the exercise, guiding each student when necessary and presenting an indicative solution, at the end.

Undergraduate lab sessions. In our undergraduate course, we supplement the theoretical lectures with eight weekly two-hour lab sessions. The first five labs focus on the rasterisation pipeline and cover topics such as transformations, illumination and shading, textures and render to textures. We exploit a modern C++ framework with OpenGL 3.3, similar to *GL-Socket* [ACFV18], that utilises live shader reloading in order to let students experiment faster with shader programming and graphics prototyping. The next two lab sessions utilise Rayground to efficiently explain the basic concepts of ray tracing. The final lab slot is reserved for assignment-related topics, such as 3D model loading and resource indexing.

In the first ray tracing lab, students get familiar with the online framework. We start by explaining the four programmable stages of the framework (see Fig. 2) and how they can be configured to generate rays and handle ray intersections with geometric objects. Then, by providing a simple template example, where rays are spawned from an orthographic camera and output the colour of the intersected object to the screen, students can experiment with the scene description by placing and parameterising the various provided geometric primitives (Fig. 3, a). As students get more confident with the mechanisms of primitive declaration, we modify the *Hit* shader and output the world coordinate normal of the object at the ray hit point (Fig. 3, b). Next, in the *Generate* shader, we convert the camera to a perspective one, using the theory and mechanisms learned from the corresponding lecture. Using live coding demonstration, by the lab assistant, a Cornell Box scene is created, illuminated from a single point light source and shaded using a Lambert BRDF (Fig. 3, c). As a task, students are requested to use the Phong model and respond to the *Miss* event by colouring the background with a gradient. A sample solution is provided at the end of the lab session.

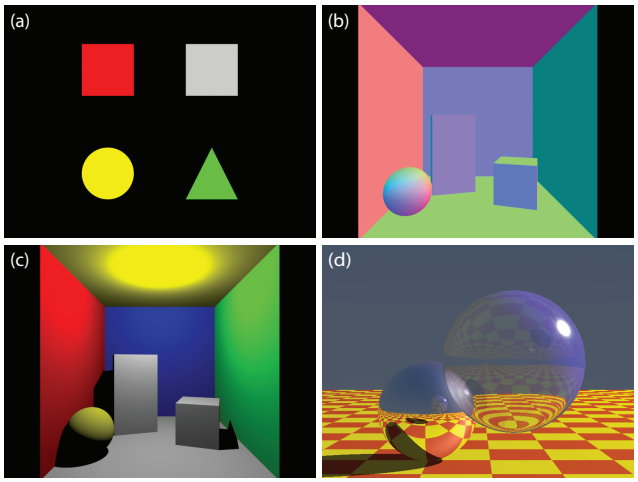

Figure 3: *Output of the first (a-c) and second (d) undergraduate lab sessions. a) Unlit shading of supported primitives. b) Cornell Box using normal vector colouring and c) Lambertian shading. d) Whitted-style ray tracer [Whi80].*

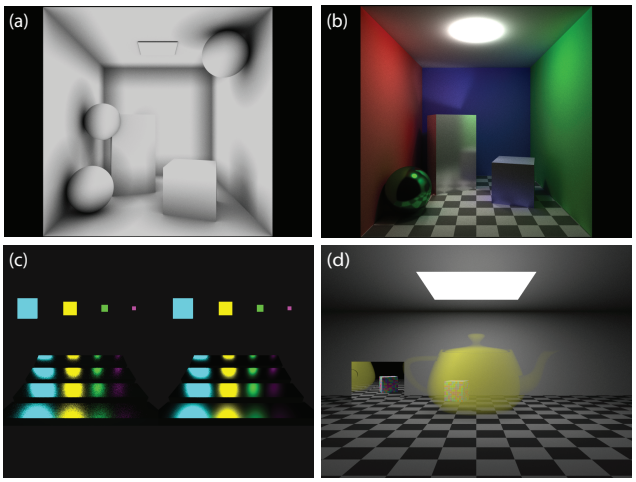

Figure 4: *Output of the graduate lab session tasks (a-d). a) Ambient occlusion. b) Unidirectional path tracer using importance sampling. c) Comparison of BRDF versus light importance sampling. d) Volumetric rendering.*

In the first part of the second lab sessions, students extend the code from the previous lab and experiment with sampling techniques. First, they perform anti-aliasing by casting rays from multiple sample locations inside each pixel. Then, instead of using a point light source to illuminate a scene, they place an area light which is sampled for visibility using area sampling. This way, they can preview soft shadows generated by the occluders of the scene. Comparative discussion with the shadow maps technique follows. After an explanation of the recursive nature of ray tracing and connection with the theory from the corresponding lecture, as a task, students create a Whitted-style ray tracer, similar to the classic scene in the paper [Whi80] (Fig. 3, d). Students that have successfully performed the task are rewarded with a bonus mark.

Graduate lab sessions. In graduate computer graphics courses, syllabi tend to cover more diverse topics than the focused introductory undergraduate ones. This means that not much effort can be spent on learning an API or development platform for ray tracing in much depth, as other subjects, such as VR-related technologies and visualisation principles and tools also occupy the available lectures and study time. In our case, a revision of basic ray tracing, light transport theory and path space sampling techniques must be covered within four three-hour lectures and the corresponding lab exercises need to follow that pace. For this reason, and considering the different background of the students, for the first two lab sessions we go over the undergraduate exercises, letting the students familiarise themselves with Rayground and the ray tracing basics. The next sessions expands on visibility determination, including ambient occlusion (Fig.4, a), as well as introduces physically-based BRDFs and path tracing with importance sampling (cosine-weighted and BRDF-driven - Fig. 4, b, c), providing the hook for the study of multiple importance sampling as a homework assignment. In the final lab sessions, we expose students to volumetric light transport (Fig. 4, d), explaining the scattering events and phase functions and use this session as a bridge to other visualisation techniques, discussed afterwards in class.

4.4. Evaluation

We evaluated both immersive lectures and the undergraduate lab sessions during the past semester. The introductory lesson to computer graphics is an elective course with an average corpus of 20 students. 15 of those students actively participated and contributed to the evaluation of the lessons. Students were given a questionnaire to fill with a series of questions that aimed to assess the usefulness and practicality of Rayground as an interactive tool for teaching computer graphics and more specifically, ray tracing. Questions had a linear rating from 1 to 5 ranging from *not at all* to *very much*.

The majority of students (80%) found Rayground very intuitive. Students quickly grasped the role of each programmable stage and, using the API, were able to quickly match previously learned concepts to the new paradigm. More than 80% commented on the very positive effect it had on their understanding of the ray tracing pipeline. The interactive nature and accessibility of the tool inspired more than 73% to further work and experiment with ray tracing in their spare time. On average, more than 80% of students welcomed the addition of Rayground as an companion tool for augmenting theoretical lessons. Also, an average of 73% of users indicated that they would rather experiment with ray tracing in the future than with rasterisation. As a side note, the concepts that troubled students the most were the recursive spawn of rays, implemented in the *Hit* stage, as well as the incremental average of the accumulation process. In conclusion, the whole experience was exciting and productive for both students and teaching assistants and we expect further improvements and better reception in future iterations of the course.

5. Discussion & Conclusions

We have introduced Rayground, a novel framework primarily triggered by the need to incorporate instructive ray tracing media in classroom and online lectures as well as to offer attractive lab assignments in an engaging manner. This work aims to set the ground for the online development of ray tracing algorithms in an accessible manner, stripping off the mechanics that get in the way of creativity and the understanding of the core concepts. Furthermore, its shader-like structure, for responding to the key events of ray generation and traversal, promotes the seamless teaching of subjects that are mostly pipeline-independent.

However, Rayground, cannot yet accommodate certain fundamental concepts of computer graphics theory limited by the current status of web-based graphics technology. Two of our major concerns include animation systems and bidirectional methods. Animation through rigid body transformations or key-frame animation is instrumental to image synthesis. Nevertheless, even desktop frameworks struggle to support animation systems that are easy to program and teach, mainly due to the high complexity required in order to maintain and expose efficient data structures in real time. Furthermore, bidirectional path tracing or photon mapping global illumination solutions are difficult to support in a simple and intuitive way with Rayground's current form. The inherent complexity of such methods would require additional components in the framework's core design. But, this would lead to a bloated framework and programming interface, as well as to a steeper learning curve for the entire system. To this end, we believe that future advances on Web-based accelerated graphics [GPU19] could shape the future research investigation on this field.

Acknowledgements

This work was supported by the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology, https://doi.org/10.13039/501100003448, under the HFRI PhD Fellowship grant with GA No. 1545.

References

[AB15] ACKERMANN P., BACH T.: Redesign of an Introductory Computer Graphics Course. In *EG 2015 - Education Papers* (2015), The Eurographics Association. 2

[ACFV18] ANDUJAR C., CHICA A., FAIRÉN M., VINACUA A.: GLSocket: A CG Plugin-based Framework for Teaching and Assessment. In *EG 2018 - Education Papers* (2018), The Eurographics Association. 2, 6

[Ang17] ANGEL E.: The Case for Teaching Computer Graphics with WebGL: A 25-Year Perspective. *IEEE Computer Graphics and Applications 37*, 2 (2017), 106–112. 2

[BSP17] BÜRGISSER B., STEINER D., PAJAROLA R.: bRenderer: A Flexible Basis for a Modern Computer Graphics Curriculum. In *EG 2017 - Education Papers* (2017), The Eurographics Association. 2

[BWF17] BALREIRA D. G., WALTER M., FELLNER D. W.: What we are teaching in Introduction to Computer Graphics. In *EG 2017 - Education Papers* (2017), The Eurographics Association. 1, 2

[Cab10] CABELLO R.: Three.js: JavaScript 3D library, 2010. 1

[CRLR14] CATUHE D., ROUSSEAU M., LAGARDE P., ROUSSET D.: Babylon.js, a 3D engine based on WebGL and Javascript, 2014. 1

[FP13] FAIRÉN M., PELECHANO N.: Introductory Graphics for Very Diverse Audiences. In *EG 2013 - Education Papers* (2013), The Eurographics Association. 2

[FWW12] FINK H., WEBER T., WIMMER M.: Teaching a Modern Graphics Pipeline Using a Shader-based Software Renderer. In *EG 2012 - Education Papers* (2012), The Eurographics Association. 2

[GPU19] GPU FOR THE WEB COMMUNITY GROUP CHARTER: WebGPU, 2019. https://gpuweb.github.io/gpuweb. 8

[JQ14] JEREMIAS P., QUILEZ I.: Shadertoy: Learn to Create Everything in a Fragment Shader. In *SIGGRAPH Asia 2014 Courses* (New York, NY, USA, 2014), SA '14, ACM, pp. 18:1–18:15. 1, 2

[KRKP*16] KLUYVER T., RAGAN-KELLEY B., PÉREZ F., GRANGER B., BUSSONNIER M., FREDERIC J., KELLEY K., HAMRICK J., GROUT J., CORLAY S., IVANOV P., AVILA D., ABDALLA S., WILLING C., DEVELOPMENT TEAM J.: Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (2016), IOS Press, pp. 87–90. 1

[KVBB*19] KELLER A., VIITANEN T., BARRÉ-BRISEBOIS C., SCHIED C., MCGUIRE M.: Are We Done with Ray Tracing? In *ACM SIGGRAPH 2019 Courses* (New York, NY, USA, 2019), SIGGRAPH '19, ACM, pp. 3:1–3:381. 1

[MG14] MÜLLER C., GÄRTNER F.: Student Project - Portable Real-Time 3D Engine. In *EG 2014 - Education Papers* (2014), The Eurographics Association. 2

[MKRE16] MWALONGO F., KRONE M., REINA G., ERTL T.: State-of-the-Art Report in Web-based Visualization. *Computer Graphics Forum 35*, 3 (2016), 553–575. 1

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A General Purpose Ray Tracing Engine. *ACM Trans. Graph. 29*, 4 (July 2010), 66:1–66:13. 1, 3

[PPGT14] PAPAGIANNAKIS G., PAPANIKOLAOU P., GREASSIDOU E., TRAHANIAS P.: glGA: an OpenGL Geometric Application Framework for a Modern, Shader-based Computer Graphics Curriculum. In *EG 2014 - Education Papers* (2014), The Eurographics Association. 2

[RME14] REINA G., MÜLLER T., ERTL T.: Incorporating Modern OpenGL into Computer Graphics Education. *IEEE Computer Graphics and Applications 34*, 4 (July 2014), 16–21. 2

[Rom13] ROMERO M.: Project-Based Learning of Advanced Computer Graphics and Interaction. In *EG 2013 - Education Papers* (2013), The Eurographics Association. 2

[RT19] RIDGE G. D., TERZOPOULOS D.: An Online Collaborative Ecosystem for Educational Computer Graphics. In *The 24th International Conference on 3D Web Technology* (New York, NY, USA, 2019), Web3D '19, ACM, pp. 1–10. 2

[SÅAM17] STRÖM J., ÅSTRÖM K., AKENINE-MÖLLER T.: Immersive Linear Algebra, 2017. www.ImmersiveMath.com. 2

[Shi19] SHIRLEY P.: Ray Tracing in One Weekend Book Series, 2019. https://raytracing.github.io. 2

[Sub18] SUBTIL N.: Introduction to Real-Time Ray Tracing with Vulkan, 2018. https://devblogs.nvidia.com/vulkan-raytracing. 1, 3

[TRK17] TOISOUL A., RUECKERT D., KAINZ B.: Accessible GLSL Shader Programming. In *EG 2017 - Education Papers* (2017), The Eurographics Association. 2

[WD15] WIENS A., DOMIK G.: In-Class Exercise for Shadow Mapping Algorithms. *IEEE Computer Graphics and Applications 35*, 3 (May 2015), 15–19. 2

[Whi80] WHITTED T.: An Improved Illumination Model for Shaded Display. *Commun. ACM 23*, 6 (June 1980), 343–349. 7

[WM19] WYMAN C., MARRS A.: Introduction to DirectX Raytracing. In *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs* (Berkeley, CA, 2019), Apress, pp. 21–47. 1, 3