Rayground: An Online Educational Tool for Ray Tracing

- "Rayground is a 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”

- "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”

- "Although ray tracing is one of the most common teaching subjects for both introductory and advanced computer graphics courses from around the world, ‘Ray tracing in one weekend’ book series is the only valuable resource available to help novice students start coding the very basics.”

- Modern low-level graphics APIs like NVIDIA OptiX, Microsoft DirectX 12 and Khronos Group Vulkan, “pose high entry barriers to students and require a very daunting and long learning process,”
Related work

- 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.

- 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.

- 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.

- 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.

- 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, shifting the focus from low-level OpenGL API to object-oriented 3D graphics frameworks.

- 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.
Related work - TL;DR

- Teaching Ray tracing has been encumbered by lack of processing power and good teaching tools until quite recently.
- Teaching frameworks have either been *too unique/abstract* in their approach, or exposed *too much* of the underlying graphics frameworks to be approachable.
- *WebGl* and *Shadertoy* have made great leaps in accessibility
  - Easy to set up, multi-platform, and has a strong student familiarity
- *ShaderLabFramework* is insufficient for properly teaching ray tracing:
  - Only two GPU ray tracing tasks were included in their course
  - It can only handle simple procedural objects that are described inside a fragment shader
Rayground

- Is an online IDE and a simple WebGL-based ray tracing pipeline.
- The pipeline is a traditional ray tracing image synthesis pipeline
  - Has multiple programmable stages via event handling shaders
The interface

```cpp
void rg_generate()
{
    // setup camera parameters
    vec3 at   = vec3(0.0, 0.0, -2.0);
    float ar = float(rg_Canvas.y) / float(rg_Canvas.x);
    float hfovRad = radians(75.0);
    float vfovRad = 2.0 * atan(tan(hfovRad/2.0)*ar);
    float vFov = vfovRad;
    float hFov = hfovRad;
    vec2 fov  = vec2(hFov, vFov);

    vec3 rendercamview = vec3(0.0, 0.0, 1.0);
    vec3 rendercamup   = vec3(0.0, 1.0, 0.0);
    vec3 horizontalAxis = vec3(1.0, 0.0, 0.0);
    vec3 verticalAxis  = cross(rendercamview, horizontalAxis);

    vec3 middle = at + rendercamview;
    vec3 horizontal = horizontalAxis * tan(fov.x * 0.5);
    vec3 vertical   = verticalAxis * tan(fov.y * 0.5);

    // jitter the position inside the pixel
    uint ctr2 = uint(rg_Pixel.x) + uint(rg_Pixel.y) * uint(1);
    vec4 randoms = rg_Random(ctr2, rg_Seed.x, rg_Seed.y);
    float sx = (rg_Pixel.x + randoms.x) / float(rg_Canvas.x);
    float sy = (rg_Pixel.y + randoms.y) / float(rg_Canvas.y);

    // generate primary rays
    vec3 pointOnPlaneOneUnitAwayFromEye = middle +
```
The ray tracing pipeline

“Since ray tracing is now tightly integrated into modern real-time rendering API, we follow a similar programming model.”
5 Stages of Rayground

Each stage are defined as a GLSL shader, each with an entrypoint function the user can implement.

- 1 declarative stage + 4 programmable stages
  - Defined in JSON and GLSL respectively
- Output from each stage is written to various variables defined per stage, shown to the left.
- The Generate and Post Process stages seem to be fragment shaders
- The hit/miss stages might possibly be compute shaders

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>All stages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vec2</td>
<td>rg_Canvas</td>
<td>canvas resolution in pixels</td>
</tr>
<tr>
<td>vec2</td>
<td>rg_Pixel</td>
<td>pixel coordinates</td>
</tr>
<tr>
<td>int</td>
<td>rg_Frame</td>
<td>frame counter</td>
</tr>
<tr>
<td><strong>Generate/Hit/Miss stages</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vec4</td>
<td>rg_PrevAccumulation</td>
<td>ray (prev) accumulation color</td>
</tr>
<tr>
<td>vec4</td>
<td>rg_PrevPayload0</td>
<td>ray (prev) payload</td>
</tr>
<tr>
<td>vec4</td>
<td>rg_PrevRayOrigin</td>
<td>ray (prev) direction</td>
</tr>
<tr>
<td>vec4</td>
<td>rg_PrevRayOrigin</td>
<td>ray (prev) origin</td>
</tr>
<tr>
<td>int</td>
<td>rg_Depth</td>
<td>ray depth</td>
</tr>
<tr>
<td>bool</td>
<td>rg_TraceOcclusion(...)</td>
<td>ray occlusion query</td>
</tr>
<tr>
<td><strong>Generate stage</strong></td>
<td>void</td>
<td>entry point signature</td>
</tr>
<tr>
<td><strong>Hit stage</strong></td>
<td>void</td>
<td>entry point signature</td>
</tr>
<tr>
<td>vec3</td>
<td>rg_Hitpoint</td>
<td>hit in world space coordinates</td>
</tr>
<tr>
<td>vec3</td>
<td>rg_Normal</td>
<td>primitive’s geometric normal</td>
</tr>
<tr>
<td>int</td>
<td>rg_MaterialID</td>
<td>primitive’s material ID</td>
</tr>
<tr>
<td>vec4</td>
<td>rg_MaterialPropertyI(...)</td>
<td>material properties, $I = [0,7]$</td>
</tr>
<tr>
<td><strong>Miss stage</strong></td>
<td>void</td>
<td>entry point signature</td>
</tr>
<tr>
<td><strong>Post Process stage</strong></td>
<td>void</td>
<td>entry point signature</td>
</tr>
<tr>
<td>vec4</td>
<td>rg_PixelColor</td>
<td>final pixel colour</td>
</tr>
<tr>
<td>vec4</td>
<td>rg_AccumulatedImage</td>
<td>2D accumulation image handle</td>
</tr>
<tr>
<td>vec4</td>
<td>rg_ImageFetch2D(...)</td>
<td>retrieve texels from 2D image</td>
</tr>
</tbody>
</table>
The scene

- Declaratively defined through a JSON schema
  - TOML would be more ergonomic, grumble grumble
- Defines the number of iterations to compute per ray
- Supports Cubes, spheres, triangles, quads
- Parsed using sajson, written in C++
  - Therefore not extendable within Rayground

```
{  
  "settings": {  
    "depth": 3  
  },  
  "objects": [  
    {  
      "type": "quad",  
      "translate": [ 1, 0, 1 ],  
      "scale": [ 2, 2, 1 ],  
      "rotate": [ 0, 1, 0, -90 ],  
      "material_property0": [ 0, 1, 0 ]  
    },  
    {  
      "type": "quad",  
      "translate": [-1, 0, 1 ],  
      "scale": [ 2, 2, 1 ],  
      "rotate": [ 0, 1, 0, 90 ],  
      "material_property0": [ 1, 0, 0 ]  
    }  
  ]
```

depth=1  depth=3  depth=30
The generate stage

- Is run per pixel *(fragment)* on the canvas
- Defines the initial rays being traced into the scene, along with some initial payload data to be passed along with them.
- Responsible for
  - Initializing the *accumulation* and *payload* variables
    - The *accumulation* is the output, usually set to black
    - The *payload* usually is the reflected color, usually set to white
  - Applying the camera transform
  - Jittering the rays within each pixels (AA)

Writes to `rg_RayDirection`, `rg_RayOrigin`, `rg_Accumulation` and `rg_Payload[0-3]`
The hit stage

- Is called when a ray intersects with something.
- Is responsible for computing the color at the intersection point.
- May trace additional occlusion rays
  - To compute things like shadows
- May then define a new ray, with some payload data to be passed along with it.

- Reads from `rg_Hitpoint` (and more), `rg_PrevAccumulation` and `rg_PrevPayload[0-3]`
- Writes to `rg_RayOrigin`, `rg_RayDirection`, `rg_Accumulation` and `rg_Payload[0-3]`
The miss stage

- Works much like the hit stage
- Is not provided with an intersection point.
- Is tasked with either ending the ray or generating a new one.
- Is free to modify the accumulation and payload variables.

- Could be used to render the skybox
The post processing stage

- **Before** this stage is run, the *accumulation* stage happens:
  - The Accumulation variable output from each trace has a RGB component and a A component:
    \[ \text{RGB}_{t+1} = \text{RGB}_t \times A_t + \text{RGB}_{t-1}(1 - A_t) \]
  - A is usually set to \( \frac{1}{rg\text{-}Frame} \), making the output the mean of all the computed frames.
  - A = 1 will bypass the accumulation stage
- This shader is fed with the final *accumulation* buffer
- Tasked with post processing the result before it is written to the canvas
  - Gamma correction
  - Warping / transformations
  - Vignetting
The starting example
What can a student do from here?

The skeleton project given to the user is very well commented, alongside with documentation for the pipeline available in a tab. The skeleton is structured in such a way that the student to quickly may start making changes:

- In minutes i had a Chromatic Aberration example working.
- Then in about 15 i had Depth of Field working.
- Other have implemented Ambient Occlusion and À-Trous Denoising.

There is a library of projects submitted by others from which you can learn.

The pipeline should to my understanding support the ability to make volumetric clouds using ray marching, and for constructing portals.
Teaching with Rayground

- The needs of the author was a ray tracing framework for graduate courses in advanced computer graphics, where the lectures addressed photorealistic rendering.
- Previous solutions gave the users troubles with general application development and debugging. Rayground reduced the barrier to entry, omitting the need to know/learn C++ at the same time.
- This have elevated the importance of ray tracing in the lectures as opposed to learning the technology stack in use. (cmake, C++, opengl, ...)
- They could therefore dedicate more class hours to teaching the basic, of simplified data acceleration structures and the comparative study of ray tracing and rasterisation.
Teaching with Rayground, cont’

- With less focus on the fiddly bits, they had time to clearly separate common topics from the specific pipeline in use, such as
  - Material properties
  - Local shading / global shading
  - Geometry representation
  - Texturing
  - Image-domain sampling
  - Aliasing and anti-aliasing

- Having a choice of multiple rendering paradigms helps the students approach the topic in a more generalizable fashion
Teaching with Rayground, cont”

● The lectures became more interactive. Demonstrations of the concepts during lectures enriched the theory presented in the static slides.
  ○ Students actively experimented using their laptops in class, further transforming the learning process into a more active and engaging one

● The ability to tweak parameters with a live feedback of the effects helped cement their understanding

● Rayground streamlined the lab course work.
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) Ambient occlusion. b) Unidirectional path tracer using importance sampling. c) Comparison of BRDF versus light importance sampling. d) Volumetric rendering.
Teaching with Rayground, cont””

- The lectures became more interactive. Demonstrations of the concepts during lectures enriched the theory presented in the static slides.
  - Students actively experimented using their laptops in class, further transforming the learning process into a more active and engaging one
- The ability to tweak parameters with a live feedback of the effects helped cement their understanding
- Rayground streamlined the lab course work.
- The students were to learn C++ in other courses, but Rayground freed up the schedule to learn C++ at a more leisurely pace.’
- The majority of students (80%) found Rayground very intuitive.
Critique

- Rayground sets an upper bound to how far a student may take their projects:
  - It’s just a static scene. You can only animate the camera and lights procedurally, but you are not able to move any objects around.
  - No user interaction. E.g. keyboard controls for the camera is impossible.

- No VR

- Nonoptimal hardware acceleration
  - WebGL
The static scene is defined using JSON:
- It is prone to syntax errors
- Has no support for comments
- **TOML is better suited for this kind of data**

There is no tool to help you import 3d meshes into the scene:

It’s not possible to combine two objects using set operators, like in povray.
Critique, cont”

- No support for textures, only procedural materials are possible.
  - No helper functions for generating noise provided, other than `rg_Random`

- Procedurally constructed or modified scenes are not possible, making these student projects impossible:
  - Procedural world generation
  - Mesh deformation / animation
  - Mesh generation
Critique, cont”

- Rayground provides very little insight into the *intersection* algorithms in use.
  - This is fine if photorealistic graphics is the only goal
  - But what if the one goal is to teach how to perform the intersection tests?
  - What if the user want so use a ray marcher instead?
Summary

- Rayground is an incredibly accessible tool for teaching ray tracing.
- It has reached the goals of its author.
- Its use case is a tad narrow. Bit it could be ‘the right tool for the job’.
- Its reduced feature set is the main factor for its effectiveness.
- Future advances in Web-based accelerated graphics could shape the future research.