Supplementary Material

We begin by discussing our implementation of VVRT. First, we review relevant features from the original Virtual Ray Tracer (VRT), before highlighting new additions in VVRT to support raycasting in Virtual Ray Tracer. Finally, we supplement some details corresponding to the evaluation.

1. Relevant Virtual Ray Tracer Features

In this section, we go over two notable features in Virtual Ray Tracer that we changed or used differently than the original.

1.1. Ray Trees

Virtual Ray Tracer uses RayTrees to visualize rays. A RayTree consists of the 0-length base ray and all its subrays. In a standalone application, this structure would not be needed for the Raycaster, since raycasting rays never reflect or refract. To avoid duplicate code with the Virtual Ray Tracer, the datatype for a ray is kept the same.

We have chosen to use the RayTrees, but not quite as intended. In the Raycaster a ray is represented as a RayTree with 4 parts: the first part is the same 0-length base ray, the second part is the ray section before entering the voxel grid, the third part is the ray section inside the voxel grid, and the last part is the ray section after the voxel grid. In this way, we can give different properties to the ray sections to help visualize them better without having to change any data types.

1.2. Trace Functions

There are two main tracing functions in the ray tracer. `Trace` is used for visualizing rays and it returns a `rayTree`. `rayTrees` can be used to draw a ray and its subrays. `TraceImage` on the other hand is used for generating a high-resolution image. It returns a color that is to be displayed on the relevant pixel. In these trace functions the main ray tracing algorithms can be found. We of course do not want to perform ray tracing, since our aim is raycasting. Therefore, these functions needed to be changed. `Trace` is replaced by `CastVisualizableRay`. `TraceImage` keeps the same name, but using inheritance it is overwritten.

The raycasting algorithm is performed in `CastRay`. `CastRay` returns a single `RCRay`, which is a child class of `RTRay` with some additions, such as a list of samples and compositing methods. Both `CastVisualizableRay` and `TraceImage` use `CastRay` to perform raycasting, then transform the returned `RCRay` into the appropriate data type.

2. Extended Classes

Because we are re-using a lot of functionality from the ray tracer, but we do not want to modify it, we decided to use inheritance. This way we can change relevant classes without changing the original ray tracing levels. There were 4 classes from the virtual ray tracer that we extended using inheritance.

- `RayManager` → `RayCasterManager`
  In this class the drawing of the rays is handled. Because we needed to draw samples as well as the rays we had to change the drawing methods.
- `RTRay` → `RCRay`
  In RTRay the information about a (sub)ray is stored. A raycasting ray has more information, like samples, that need to be stored. For this we made RCRay.
- `UnityRayTracer` → `UnityRayCaster`
  In this class the ray tracing/casting algorithms are implemented. Since our aim is to perform raycasting instead of ray tracing we overwrite the trace methods.
- `RayTracerProperties` → `RayCasterProperties`
  In `RayTracerProperties` the control panel input and output is handled. Since the Raycaster has some overlapping controls, but also some new ones, we made a child class. We then added the new controls to the child class `RayCasterProperties`, and we deactivated any controls that we did not want to use in the unity editor.

3. New Classes

New classes have also been added to the application. These classes are only related to raycasting and are not tied to the ray tracing levels in any way.

- `VoxelGrid`
  To represent our voxel grid data in Unity we made a prefab. A prefab in Unity is a template with properties and settings that can be reused in multiple scenes. To this prefab, we added a transparent cube object to show in the scene. We also added the class `VoxelGrid` to represent our data.
- `RayCalculationBreakdown`
  This class handles the Ray Calculation Window. Its variable breakdown part is static and is simply a text box. The formulas are 4 pre-made images that are saved as sprites. When the compositing method is changed the relevant formula image is displayed.
- `Sample`, `SampleObject`, `SampleObjectPool`, and `SampleRenderer`
  These four classes together handle the samples. `Sample` itself is the data of a sample. `SampleObject` is a sample object in Unity space. `SampleRenderer` is tied to a sample object and draws it. `SampleObjectPool` is used for optimization.
  There are many of these samples at a given time and to keep the application running smoothly we use the object pool. We start with an initial number of 128 samples. These samples are loaded, but set to inactive and therefore invisible. If more than 128 samples are needed they are created, but not destroyed, so they can be reused as well. The `SampleObjectPool` handles activating, deactivating and creating new samples.

4. 3D Texture Generation

To create a 3D texture for each voxel grid, we created a utility script for developers of VRT to turn any new voxel grid file into a corresponding 3D texture. It can be run from within the Unity development environment without running play mode; see Fig. 1.

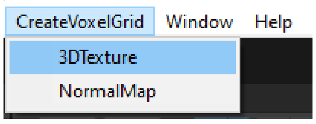

Figure 1: *The script can be executed from the Editor*

There is also a script in place to store pre-computed normals for voxel grids, facilitating more efficient lighting calculations in the future, but it is not yet in use.
First, the developer needs to manually adjust the dimensions of the voxel grid that is to be processed. Next, they input the file path of the '.raw' file in which the voxel grid for Virtual Raycaster is stored.
Upon startup, the script then creates a new 3D texture and loops through each texel, saving the density from the '.raw' file in the alpha channel of the texel's color. Changes to the texture are then applied and the new Asset is saved in the Resources folder with a file name specified by the developer.

An auxiliary script is attached to the voxel grid object, which enables the generation of the main camera's depth texture, needed for depth testing and correct blending of the volume and geometric objects.

5. Raycasting Shader

5.1. Shader Properties

The shader's properties are its variables which are exposed to the Unity environment and can be edited by other scripts, providing an endpoint where synchronized variables can be passed along.

- `_MainTex`: The 3D texture containing the density values for raycasting.
- `_MinDensity`: A threshold below which density values are ignored.
- `_StepSize`: The distance between each sample along the ray.
- `_AlphaCutoff`: Maximum allowed opacity for compositing.
- `_CompositingFunction`: Determines which compositing method is used (0 for Accumulate, 1 for Maximum, 2 for Average, 3 for First).
- `_TargetDens`: Target density for the first compositing method.
- `_Transfer1` to `_Transfer5`: Lookup table parameters for color transfer.
- `_Transfer1c` to `_Transfer5c`: Color values used for mapping densities to colors.

5.2. Blending Operations

The shader includes some settings to ensure that the voxel grid blends correctly with other geometry:

- `'Queue' = 'AlphaTest'` puts the object at position 2450 in Unity's render pipeline, after all opaque objects, to ensure that they are in the depth buffer before the shader executes.
- `ZWrite Off` disables writing to the depth buffer, to treat it as a transparent object.
- `ZTest LEqual` sets built-in depth testing conditions. (This is separate from the custom depth test described above for volume/geometry blending.)
- `Blend SrcAlpha OneMinusSrcAlpha` Sets the combination of the output of the fragment shader with the render target to premultiplied transparency.

5.3. Vertex Shader

The vertex shader performs three tasks:

- Prepares the object vertex by converting it to object space, in order to set a starting point for our ray marching ray.
- Calculates the unit vector from the camera to the object, which will determine the direction of the ray in the fragment shader.
- Calculates the vertex's clip position, which is needed to get the screen UV coordinates to sample the depth texture.

5.4. Central Differences

To compute normals for lighting, the `centralDiff` method computes a vector composed of the differences in density between voxels along all axes, given a delta value which determines the distance from the point for which the normal is to be estimated. At this distance, in both directions, samples are taken and the two samples are subtracted from each other. As for each axis such a difference is computed, the three differences together create a vector which serves as a surface normal.

This means that, mathematically speaking, the gradient of the scalar field is approximated using the following central differences formula for each axis:

$$\mathbf{N} = \begin{bmatrix} \rho(x+\Delta,y,z) - \rho(x-\Delta,y,z) \\ \rho(x,y+\Delta,z) - \rho(x,y-\Delta,z) \\ \rho(x,y,z+\Delta) - \rho(x,y,z-\Delta), \end{bmatrix} \tag{1}$$

where:

- ρ is a function representing the sampling of density from the voxel grid;
- Δ is a given offset.

Once the normal vector is calculated, it needs to be normalized (to a length of 1) before it can be used for illumination.

5.5. Fragment Shader

The first meaningful calculation the fragment shader performs is determining the number of samples taken per ray, by taking the step size (the distance between each sample, given by the user in terms of voxels, then converted to Unity units) and dividing the diagonal of (and therefore longest distance across) the voxel grid by it. After this step count is established, a loop runs through each sample position, samples the 3D texture to get a corresponding density value and applies the selected compositing function.
Afterwards, the Transfer function is applied to output the desired color.

6. Controls

We expanded the control panel of the ray tracer with new settings and hid some controls that are irrelevant or incompatible with raycasting. The controls that the Ray Tracer and Raycaster do share are:

- Hide no hit rays
- Animate rays
- Animate sequentially
- Loop animation
- Animation speed
- Supersampling
- Supersampling animation
- Render image
- Open image
- Fly to virtual camera

Some of the new controls have already been mentioned: selecting a voxel grid, selecting a compositing method, and changing the transfer function. These are the bare controls that are needed for raycasting. To allow for experimenting with the application and different visualization styles we added four more settings described below.

7. Lighting

Phong illumination is a model used to simulate the way light interacts with surfaces to create illumination effects. It involves calculating three types of light reflection: ambient, diffuse, and specular. Below is a brief breakdown of each component:

- **Ambient Reflection:**
  - Represents the constant, non-directional light present in the scene, ensuring that objects are not completely dark even if they are not directly illuminated.
  - Calculated as:

    $$I_{\text{ambient}} = k_{\text{ambient}} \cdot I_{\text{ambient}},$$

    where k_{ambient} is the ambient reflection of the surface, and I_{ambient} is the intensity of ambient light.
- **Diffuse Reflection:**
  - Models the scattering of light in all directions when it hits a rough surface. The intensity varies with the angle between the light vector and the surface normal.
  - Calculated using Lambert's cosine law:

    $$I_{\text{diffuse}} = k_{\text{diffuse}} \cdot I_{\text{light}} \cdot \max(0, \mathbf{L} \cdot \mathbf{N}),$$

    where k_{diffuse} is the diffuse reflection, I_{light} is the luminosity of the light, $\mathbf{L}$ is the light vector, and $\mathbf{N}$ is the normal vector of the surface, both normalised.
- **Specular Reflection:**
  - Simulates the bright spots of light that appear on shiny surfaces where the light is reflected directly to the viewer. It depends on the viewer's angle and the light reflection direction.
  - Calculated using the Phong reflection model:

    $$I_{\text{specular}} = k_{\text{specular}} \cdot I_{\text{light}} \cdot (\max(0, \mathbf{R} \cdot \mathbf{V}))^{\alpha},$$

    where k_{specular} is the specular reflectance, $\mathbf{R}$ is the reflection vector of the light, $\mathbf{V}$ is the view direction, and α is the shininess exponent controlling the size of the specular highlight.
- **Final Color Calculation:**
  - The final color of a point is the sum of ambient, diffuse, and specular contributions:

    $$I_{\text{final}} = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}.$$

CPU	Graphics Card	Avg. FPS
Ryzen 5950x	RTX 3080	360[R]
Ryzen 5800x3D	RTX 3080	144[R]

Table 1: *Frames per second average at default settings.*

Type	File Size	Ryzen 3700x, RTX 2070	Ryzen 5950x, RTX 3080
Bucky	0.032MB	0.30s	0.11
Bunny	92MB	6.99s	1.96
Engine	16MB	4.23s	0.69
Hazelnut	131M	17.08s	3.45

Table 2: *Loading times in seconds for different data and hardware.*

8. Live Raycasting User Study Instructions

We provided participants with the following instructions:

Open **Virtual Ray Tracer.exe**, select **Levels**, then **15. Ray Casting**.

Follow the Tutorial on the bottom left of the screen twice:

1. On the first playthrough, **disable** the "Show Preview" function (at the very bottom of the right menu, as explained in the tutorial).
2. On the second, leave it enabled and see how it changes your experience.

Next, fill out this questionnaire.

9. Performance

Our measurements show that with modern hardware we even yield frame rates hitting the monitor's refresh rate (marked with an [R] in Table 1). Frame rates were tested with the Bunny dataset at a sample distance corresponding the size of a voxel, all other settings were left at their default values. The loading times of different datasets are listed in Table 2. While this can take up to several seconds depending on hardware and data, it can still be considered fast enough for a user to do this during an exploration session.

10. User Study Comments: Raycasting Process

- "As non-experts, we don't yet fully understand what we're doing. Perhaps start by explaining the purpose and what it aims to achieve. What are we working toward, and what is the end goal?"
- "Render preview pixels only became clear to us in the final step."

- "I quickly played around with the application and found it quite interesting! [...] Here are some suggestions [...]:
  - It would be nice to see the voxel grid (bucky, bunny, etc.) in some way inside of the unit cube [...]
  - The 5 parameters below the plot of the transfer function don't have labels. It's better to indicate what they're for exactly.
  - Is "bucky" the same dataset as the CUDA volume renderer example uses? If so, then such an abstract dataset may not be a good initial choice. [...]"
- "Great addition! I think you could make it even better by having a couple of "scenarios", where you tell the user how to set up the ray caster. E.g. only visualize the surface, or only visualize the most dense parts of a volume. One feature you could also consider is to visualize the voxel grid in some way, so you can see where the rays sample the volume (this is a nice to have)."

Here we see the request to create the live raycasting feature which has since been implemented. Additional suggestions are clearer explanations of the goal of ray casting, as well as creating multiple levels where the user will aim to visualize different aspects of a dataset. Overall the comments were positive with ideas for future improvements and features.