Deferred Image-based Ray Tracing
Demo Documentation

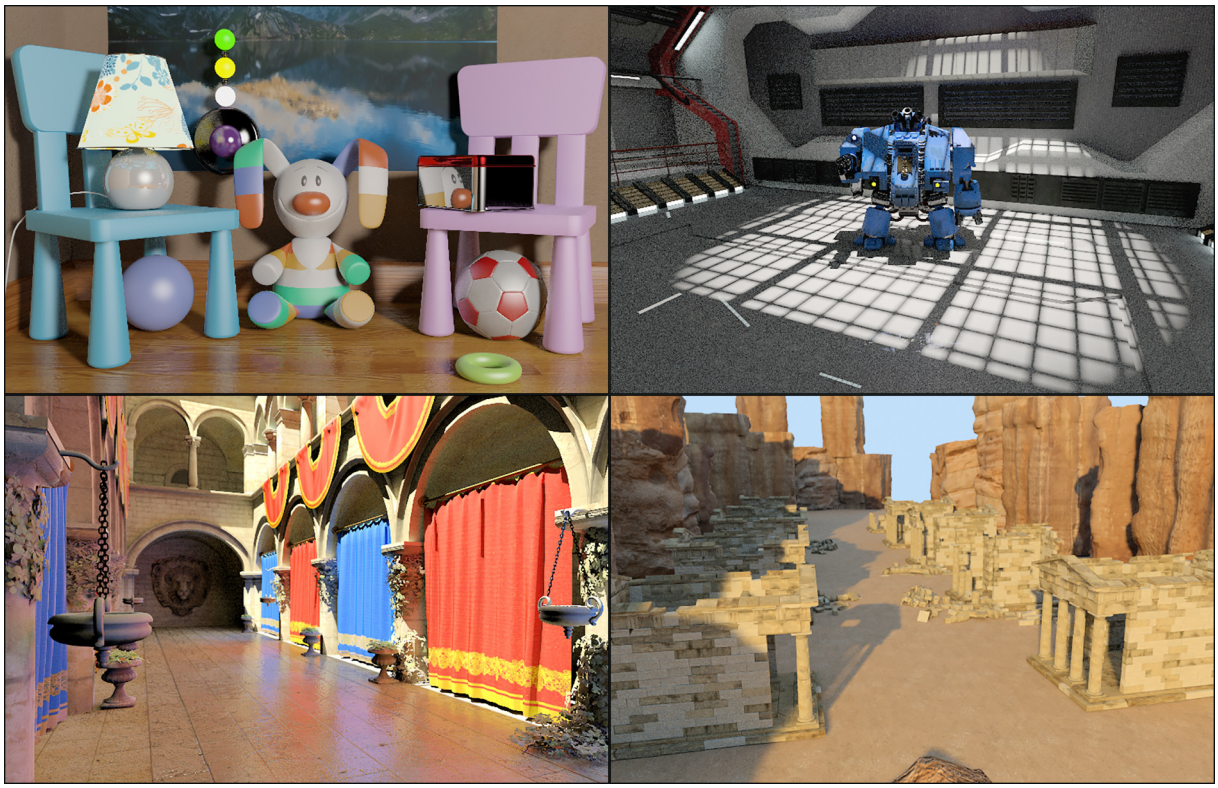

Figure 1: Resulting images of the scenes available in this demo, as they have been rendered with DIRT.

Introduction

This demo is a prototype implementation of the paper *Deferred Image-based Ray Tracing* (DIRT) [3], a rasterization-based ray tracing pipeline able to support: (i) full global illumination algorithms such as path tracing, (ii) quality identical to the traditional ray tracing based on spatial acceleration data structures, since ray tracing routines are based on ray-primitive intersections rather than ray-fragment ones, (iii) reduced memory requirements compared to traditional rasterization-based techniques (see the demo of the corresponding paper [5] on the Computer Graphics Group's website of the Athens University of Economics and Business [2]) as the acceleration data structure (ADS) stores primitive ids rather than complete per-fragment data, and finally (iv) fully dynamic environments as the DIRT's acceleration data structure is rebuilt in every frame.

The current demo is implemented using OpenGL 4.4 in our custom graphics engine, XEngine [4] (an older version is available here) and has been tested on NVIDIA GPUs with Maxwell architecture or newer. It encompasses several performance and quality bug fixes compared to the original paper (e.g., a smaller number of depth subdivisions is required now for optimal results, further reducing the memory requirements, improved BSDF sampling routines, etc.).

For any suggestions regarding the demo, e.g., bug fixes, performance improvements, optimizations, etc., or this document, e-mail me at kvardis@aueb.gr, kvardis@hotmail.com, or message me on my Twitter account @kvarcg.

Fast Details about the Demo

The main purpose of this demo is to show provide an introduction to primitive-based screen-space ray tracing and how it operates under various conditions. It encompasses a much larger degree of

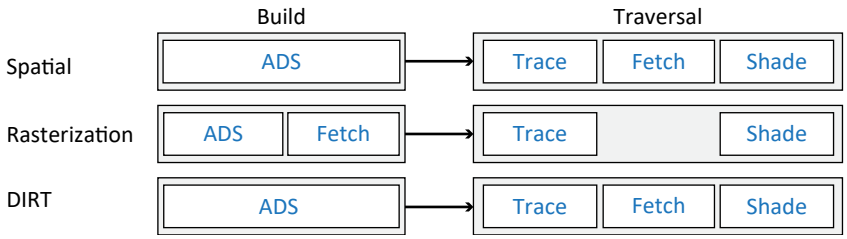

Figure 2: Tracing pipelines overview.

functionality than a product-ready version should have since most of its internal parameters can be altered, e.g., hierarchical tracing mip levels, depth subdivisions, tiling, ray types, etc. As such, the underlying implementation is unavoidably "heavier" than it should be in terms of both performance and memory, since it must manage various additional buffers and perform extra computations for all its visualization and testing purposes. The resulting images are those of a raw path tracing implementation based on simple BSDF importance sampling (see Figure 1). There is no filtering applied to the final result. The demonstrated scenes were designed with real-case scenarios in mind and with no regard as to any "optimizations" that improve performance on certain cases, such as on the number of primitives, the placement of the objects in the virtual world, the number and type of materials and/or textures they contain.

Fast Details about DIRT

DIRT performs ray tracing by applying the deferred tracing scheme of spatial-based methods in a rasterization-based framework: (i) an ADS is constructed containing primitive-based scene information required only for traversal, and (ii) ray traversal is performed iteratively for each path event, following a Trace-Fetch-Shade approach to compute global illumination (i.e., direct and indirect paths) (see Figure 2).

The construction stage of the ADS is responsible for generating and storing clipped primitives in screen-space tiles in the form of a linked-list. Essentially, the underlying data structure is a view-based (single- or multi-view) perspective irregular grid. A tile is represented in the image-domain by a 1×1 pixel block or higher (2×2, 4×4) (see Figure 3). In the depth-domain, the primitives of each tile are also stored in uniformly subdivided depth bins where the depth extents of each tile are associated with the range of the first and last clipped primitives encountered in depth space rather than the near/far extents of the entire view (see Figure 5(a,b)). The ADS also creates a mipped min-max *depth bounds texture*, necessary for the generation and placement of the clipped primitives into their associated depth intervals, as well as for performing hierarchical screen-space ray tracing. To ensure that all primitives are captured conservative rasterization is employed. Note, however, that this approach does not capture zero-area primitives, an issue that can be remedied by slightly shifting these primitives during the geometry shader stage.

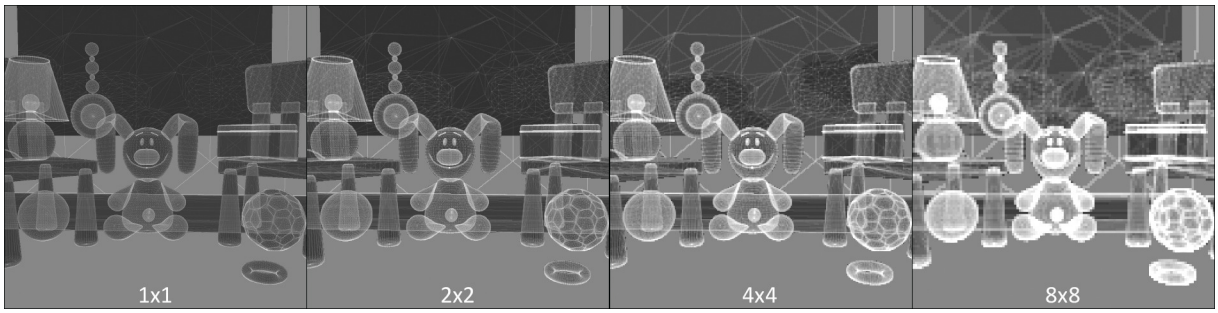

Figure 3: Different screen-space tiles representations. Changing the tile size trades traversal overhead in image-space with larger lists in the depth domain and reduces construction times and memory requirements due to the lower fragment generation. In practice, a tile size of 2×2 or 4×4 performs best in most scenarios.

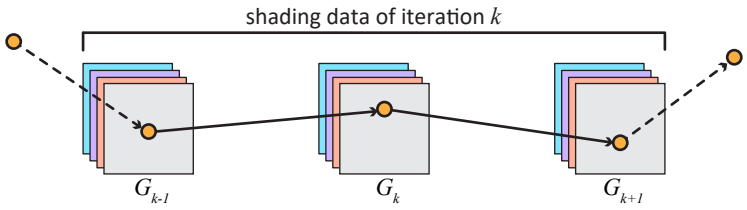

Figure 4: To compute the illumination for point G_k, the shading buffer contains information only for the three points contributing to the current event iteration k.

The tracing stage is executed in an iterative manner, requiring three passes for each path event: a *Trace*, *Fetch* and *Shade* pass. The key idea here is that during each event k, per-pixel material information for the three required points (G_{k-1}, G_k, G_{k+1}) that contribute to the illumination calculations of k are stored in an auxiliary shading buffer instead of the ADS (see Figure 4). Since the shading buffer is resolution-dependent and memory-bound (3 times the viewport resolution), the memory requirements are significantly reduced compared to traditional fragment-based rasterization methods that store the entire material data as part of the ADS (i.e., scene-dependent storage of material information).

During traversal, rays are generated based on the current point G_k (*Trace* pass) and traversed through the scene, storing any identified hits at the intersection (pixel) location of a *hit buffer*, whose memory requirements are also bound and resolution-dependent. All identified hits are then fetched during the *Fetch* pass, through a traditional rasterization step, and stored back at location G_{k+1} of the shading buffer. The illumination is then computed in the final pass, followed by a left shift operation to set the identified hit as the starting location of the ray for the next event (*Shade pass*). For example, a 2-bounce indirect illumination would require 3 iterations ($k = 3$). An interesting benefit of employing the rasterization pipeline is that an optional *Direct Visibility* pass can be performed right after the ADS construction to store the direct illumination data in a very efficient manner, dispensing with the tracing procedure and reducing the total iterations to 2.

Naive image- and depth-space (i.e., multi-fragment) ray tracing is very inefficient (see Section 5 in the paper). DIRT significantly increases the traversal performance by exploiting empty-space skipping strategies such as: (i) hierarchical screen-space ray tracing (see Figure 6) and (ii), skipping depth bins or even entire tiles that are not crossed by a ray [5] (see Figure 5(c)). Once a set of appropriate bins has been found, traditional ray-primitive intersection testing takes place to identify any potential hits.

DIRT supports two ways of capturing the scene and performing ray-tracing: (i) a multi-view structure, capable of high-quality and accurate results at interactive rates, and (ii) a single-view structure, able to maintain real-time frame rates while providing correct global illumination restricted to the geometry within the view.

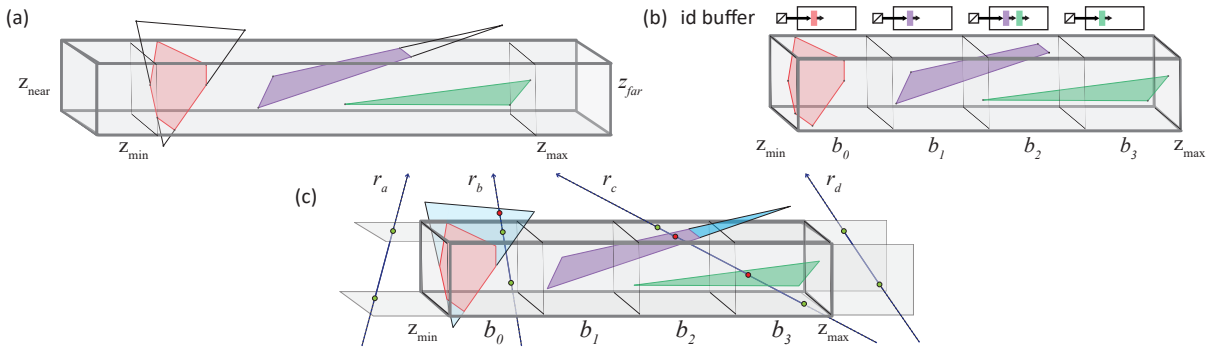

Figure 5: Per-pixel primitive clipping is exploited for correct construction including (a) depth range computation and (b) multi-bucket placement. (c) During tracing, empty-space skipping is performed when rays pass outside depth boundaries (r_a, r_d) or when a ray intersects with a subset of the depth bins within the tile of interest (r_b). A valid hit is found when a ray intersects a primitive within the bucket (r_c).

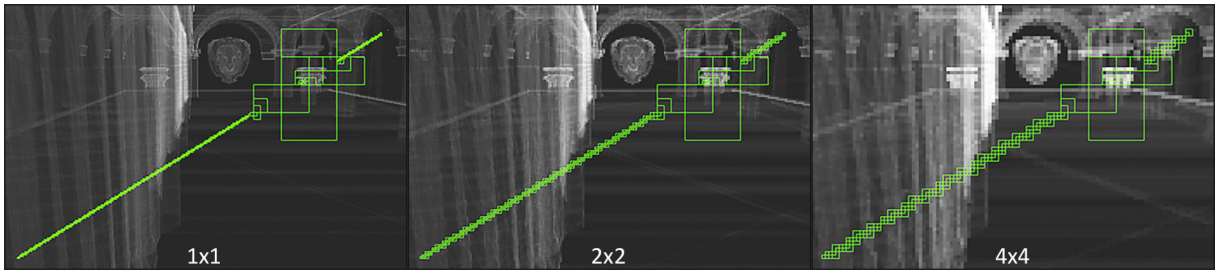

Figure 6: Tile-based hierarchical tracing in screen-space. Rays travel from left to right. Depending on the tile size, fewer checks are made in screen-space as the tile size increases.

Demo Details

Requirements

The demo requirements are:

- Windows 64-bit operating system
- OpenGL 4.4
- Visual C++ 2015 x64 Update 3 redistributable (in the redist folder)
- NVIDIA GPU with Maxwell architecture or newer since the current implementation uses conservative rasterization through the *GL_NV_conservative_raster* extension

To Execute The Demo

The demo contains 5 different scenes of arbitrary complexity. To run a scene simply double-click on its associated bat file. For example:

- *"run_bunny_day_1024_512_full.bat"* executes the *bunny_day* scene at 1024×512 resolution using the "full" DIRT version, i.e., the multi-view version.
- *"run_bunny_day_1024_512_single.bat"* executes the *bunny_day* scene at 1024×512 resolution using the single-view implementation of DIRT.

To Run Your Own Scenes

- Create a new folder containing your .obj files in the EngineData folder.
- Create a new .scene file, by copying the *template_dirt_<views>* files and edit the parameters in the new scene file using a text editor.
- Edit the following parameters in the .mtl files of your obj to support XEngine's internal material system:
  - There is no notion of specular color and *Ks* is used to represent scalar reflectivity. Therefore for a reflectivity of 0.1 use *Ks 0.1 0.1 0.1*.
  - Use *metallic*, in the range of [0-1], to set a material as metallic.
- Create a new .bat file containing the following:

```
start Demo.exe <your_scene>.scene <width> <height> 0 <"Window Title">
exit
```

- Run the new file.

Notes on First Time Running the Demo

XEngine supports dynamic object loading and a binary representation (*.sbin files) of .obj files to increase responsiveness and improve loading times. The binary files are generated the first time an .obj file is loaded and a warning message will show up indicating that the generation will take place. The scene however (even an empty one) will be functional right from the beginning and will show each object as soon as it becomes ready for rendering.

To disable binary file functionality, set the XML attribute *binary_loading* of the *world* node at the top of the corresponding scene file to *false*. To remove those files at any time, simply delete all *.sbin files in the *EngineData* folder. Finally, to disable dynamic loading, set the *multithreaded_loading* of the *world* node to *false*.

Current Limitations

The current implementation is limited in the following:

- The cubemap version of this demo uses 7 views instead of 6. The additional view is essentially the camera view that is employed for two reasons: i) it allows the support of the single-view version in the same implementation, and (ii) it provides a small performance benefit (at the expense of the extra memory cost) in the cubemap version for the direct/1-bounce indirect calculations as minimal view-switching is required.
- The current implementation of hierarchical tracing currently supports resolutions in powers of two (e.g., 512×512, 1024×512, 1024×1024, etc.). Window sizes are adjusted this way and therefore, resize is not available. Linear tracing, however, is not limited by this.
- Only diffuse/glossy/specular/emissive materials are supported (no transmission).
- Only one point spotlight source is supported.
- Zero-area primitives are not handled.

General Tips for Best Performance/Memory balance

DIRT has only two parameters that affect its performance/memory: tiling and depth subdivisions.

- Tiling should be between 2×2 and 4×4 for most scenes/resolutions provided in the demo. This can be altered by changing the XML attribute *block_size* of the *tracing* element in the associated scene file to the previous/next power of 2. Decreasing the tile size also increases the memory requirements.
- Depth subdivisions should be between 16 and 40 for most scenes. This can be altered by changing the XML attribute *buckets* of the *tracing* element in the associated scene file. Increasing the depth subdivisions also increases the memory requirements.

Troubleshooting

- Check the log file in the Logs folder for any *Warning* or *Error* messages.
- On GPUs, there may be the case where construction times are extremely high (>100ms) due to very large fragment generation (e.g., on the *Temples* scene in 2K resolution on NVIDIA GTX 970). In this case try increasing the tile size either by using the key binding or (better alternative) by changing the XML attribute *block_size* in the associated scene to the next power of 2 (e.g., change 2 to 4).
- If tracing is very slow on your GPU, run a lower resolution version or the single-view variation of the scene.

Key Bindings

An extensive list of key bindings is provided for information and testing purposes (see also Figure 7):

Tracing Settings	
M:	Enable/Disable progressive rendering
3/4:	Increase/Decrease number of light bounces, e.g., 1 is direct lighting, 2 is direct + 1 indirect bounce, etc.
F:	Enable/Disable Depth of Field
C/V:	Change the aperture size to increase/decrease Depth of Field
Z/X:	Increase/Decrease the focus distance, i.e., where the objects appear sharp
Testing Settings	
G:	Changes the View mode, i.e. the number of views between single and cubemap
5/6:	Increase/Decrease number of depth bins. i.e., the number of per-pixel depth subdivisions
7/8:	Increase/Decrease Tile Size, i.e. alters the lower (i.e., high-detailed) screen-space rasterization resolution during the ADS construction
9/0:	Increase/Decrease hierarchical tracing highest mip-level (i.e., low-detailed) one. This can be used in conjuction with the Ray Preview mode to visualize how the rays traverse the scene.
Q:	Changes visibility calculations (raytraced/shadow maps). The raytraced implementation is currently suboptimal.
-:	Enable/Disable Ray Preview mode. This causes the ray of the central pixel to be written to a debug ray texture and previewed in Preview mode.
=:	Changes the rays to reflect different materials. Ray types cycle between: Default (no change), diffuse, glossy (roughness=0.1), specular and visibility.
Engine Settings	
T:	Enable/Disable GPU timers
F8:	Show/Hide screen text
P:	Preview internal buffers
1/2:	Prev/Next preview buffer
F5:	Screenshot capture of the current preview buffer
F6:	Export camera and user details, such as position, target, direction, near and far field, in the Log file
R:	Reloads the shaders
ESC:	Exit the application
Camera Movement	
Arrow keys:	User movement on camera UW(XZ) axis
PgUp/PgDn:	User movement on camera V(Y) axis
Mouse(Left Button):	Rotation on UV(XY) axis

General Notes Regarding Quality/Performance/Memory/UI

- The actual memory allocations are higher than needed (see Memory in Figure 7) (i) to accommodate both single-/multi-view implementations (e.g., 7 views instead of 6), (ii) to provide a testing/visualization environment (see Test Settings in Figure 7) and, (iii) due to functionality limitations in OpenGL. These have also an impact on the performance of the current implementation.
- The single-view versions do not use the background color for any rays exiting the view. Therefore, the outdoor scenes appear darker in their single-view versions.
- Use GPU timers to see the exact construction/tracing times. Since all displayed timers are measured synchronously, disable them for better performance.
- Switching between single/cubemap view is not available in the single-view demo, as it requires reallocation of all the internal buffers.
- Tile size cannot be lower than the initial state, as it requires reallocation of all the internal buffers. The effect of changing the tile size can be seen in the “Preview Buffers” mode.

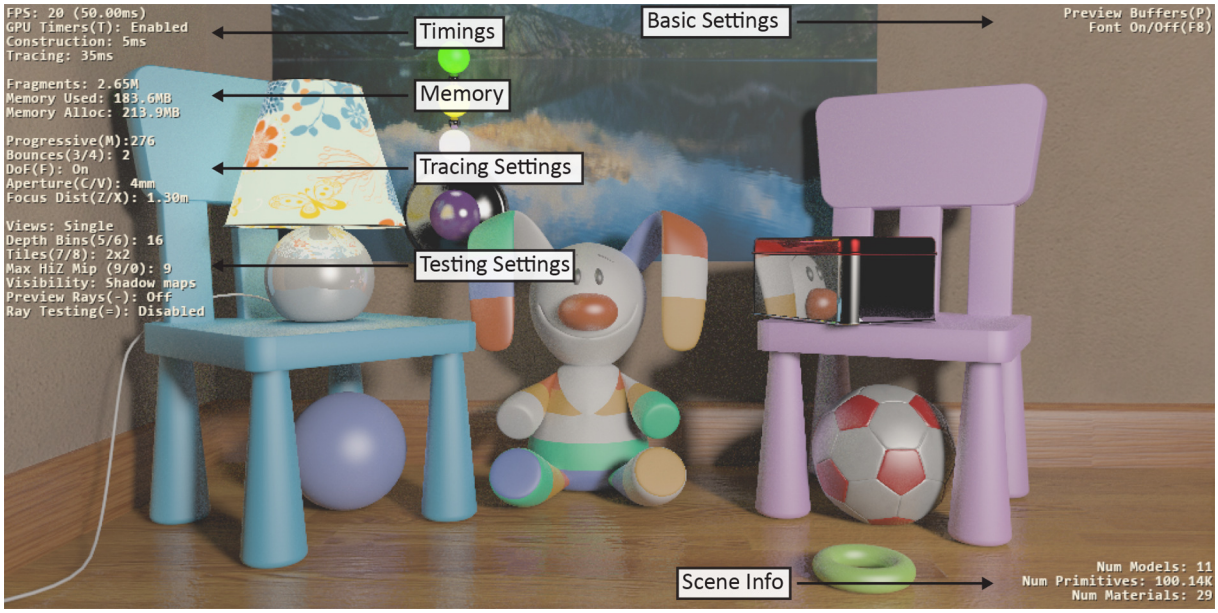

Figure 7: The screen text displays information relevant to memory, performance and tracing as well as various visualization/testing settings.

- Depth bins cannot be larger than the initial state, for the same reason as the tile size.
- Ray Preview should be used for visualization purposes only as it imposes a noticeable performance overhead during tracing due to the extra write operations in the ray preview buffer.
- The different ray types are demonstrated on the *indirect* paths as direct shading is very fast. They should only be used for benchmarking and visualization purposes. When visibility rays are selected light bounces is set to 2 (1 indirect).
- Maximum number of light bounces is 3.

GLSL Source Code Reference

Example Source Code

Example source code (i.e., no path tracing implementation) is contained in the *Example Shader Code* folder. This is the sample shader code that was provided as supplemental material in the original paper.

ADS Construction Stage

The shader source code for the ADS construction stage is located at *Shaders\Tracing\PrimitiveConstruction* and contains the following files:

- *DIRT_Clear*: Screen-space pass to clear the internal data structures.
- *DIRT_FillDepth*: Geometry pass to computes the min/max depth bounds for each pixel by clipping against the incoming primitives. The (explicit) vertex buffer is also filled here, during the geometry shader invocation.
- *DIRT_FillDepthMipmap*: Computes a mipped representation of the depth bounds, used for tiling and hierarchical tracing.

- *DIRT_FillPrimitives*: Geometry pass to construct the id buffer. The primitives are also clipped here to be stored in their corresponding depth bins.
- *DIRT_DirectVisibility*: The (optional) direct rendering pass. Stores shading properties for direct visibility in the shading buffer. If this is executed, a Shade pass is also executed before entering the Traversal stage in order to compute the direct illumination. If camera shading effects are present, this should be skipped. This pass must not use conservative rasterization.
- *DIRT_Resolve*: Screen-space pass used for previewing the DIRT ADS.

Traversal Stage

The Traversal stage consists of the 3 passes required for ray tracing: Trace, Fetch and Shade and is executed iteratively, for each path event k. The shader source code is located at *Shaders\Tracing\PrimitiveTracing* and contains the following files:

- *DIRT_Trace*: Performs single- and multi-view screen space hierarchical traversal, id buffer lookup, depth bin empty space-skipping and analytic intersection tests. All hits are stored in the hit buffer at the pixel hit location. Tracing starts from point G_k.
- *DIRT_Fetch*: Fetches shading properties for all hits stored in the hit buffer. The final information is stored in the shading buffer at location G_{k+1}.
- *DIRT_Shade*: Computes the lighting contributions for the current path event for the shading buffer points G_{k-1}, G_k, G_{k+1}. A final left shift operation is performed in the shading buffer contents if another iteration is pending.
- *DIRT_DepthMask*: Screen-space optimization pass to create the mask texture, flagging inactive pixels. This is exploited during tracing and shading by enabling Early Fragment Testing.
- *DIRT_Preview*: Screen-space pass used for (i) combining an ADS view with the ray preview, and (ii) viewing all ADS views along with the final result in the same preview window.

Helper GLSL Files

The shader source code for all included files is located at *Shaders\Include\DIRT* and contains the following files:

- *DIRT_data_structs.glsl*: Data structs definitions and associated packing functions.
- *DIRT_basic_lib.glsl*: Basic functionality, e.g., random number generators, normal compression routines, etc.
- *DIRT_vertex.glsl*: Vertex data construction.
- *DIRT_clip.glsl*: Primitive clipping.
- *DIRT_tracing.glsl*: Single and multi-view hierarchical screen-space ray tracing.
- *DIRT_abuffer_cubemap.glsl*: Per-pixel linked-list tracing, depth-based skipping and ray-triangle intersections.
- *DIRT_lighting.glsl*: Path tracing illumination and shadow mapping routines.

Content Credits

- The Sponza Atrium model was downloaded from Morgan McGuire's Computer Graphics Archive [1].
- The models comprising the Bunny, Hangar and Temples scenes were created by Prof. Georgios Papaioannou at the Computer Graphics Group of the Athens University of Economics and Business [2].

References

[1] M. McGuire. *Computer Graphics Archive*. July 2017. URL: https://casual-effects.com/data (Cited on page 9).

[2] G. Papaioannou. *Computer Graphics Group*. URL: http://graphics.cs.aueb.gr/graphics/downloads.html (Cited on pages 1, 9).

[3] K. Vardis, A. A. Vasilakis, and G. Papaioannou. *DIRT: Deferred Image-based Ray Tracing*. In: *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. Ed. by U. Assarsson and W. Hunt. The Eurographics Association, 2016. ISBN: 978-3-03868-008-6. DOI: 10.2312/hpg.20161193 (Cited on page 1).

[4] K. Vardis and G. Papaioannou. *XEngine*. URL: http://graphics.cs.aueb.gr/graphics/downloads_tutorials.html (Cited on page 1).

[5] K. Vardis, A. A. Vasilakis, and G. Papaioannou. *A Multiview and Multilayer Approach for Interactive Ray Tracing*. In: *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '16. Redmond, Washington: ACM, 2016, pp. 171–178. ISBN: 978-1-4503-4043-4. DOI: 10.1145/2856400.2856401 (Cited on pages 1, 3).