Run-Time Code Generation for Materials

Stephan Reiter

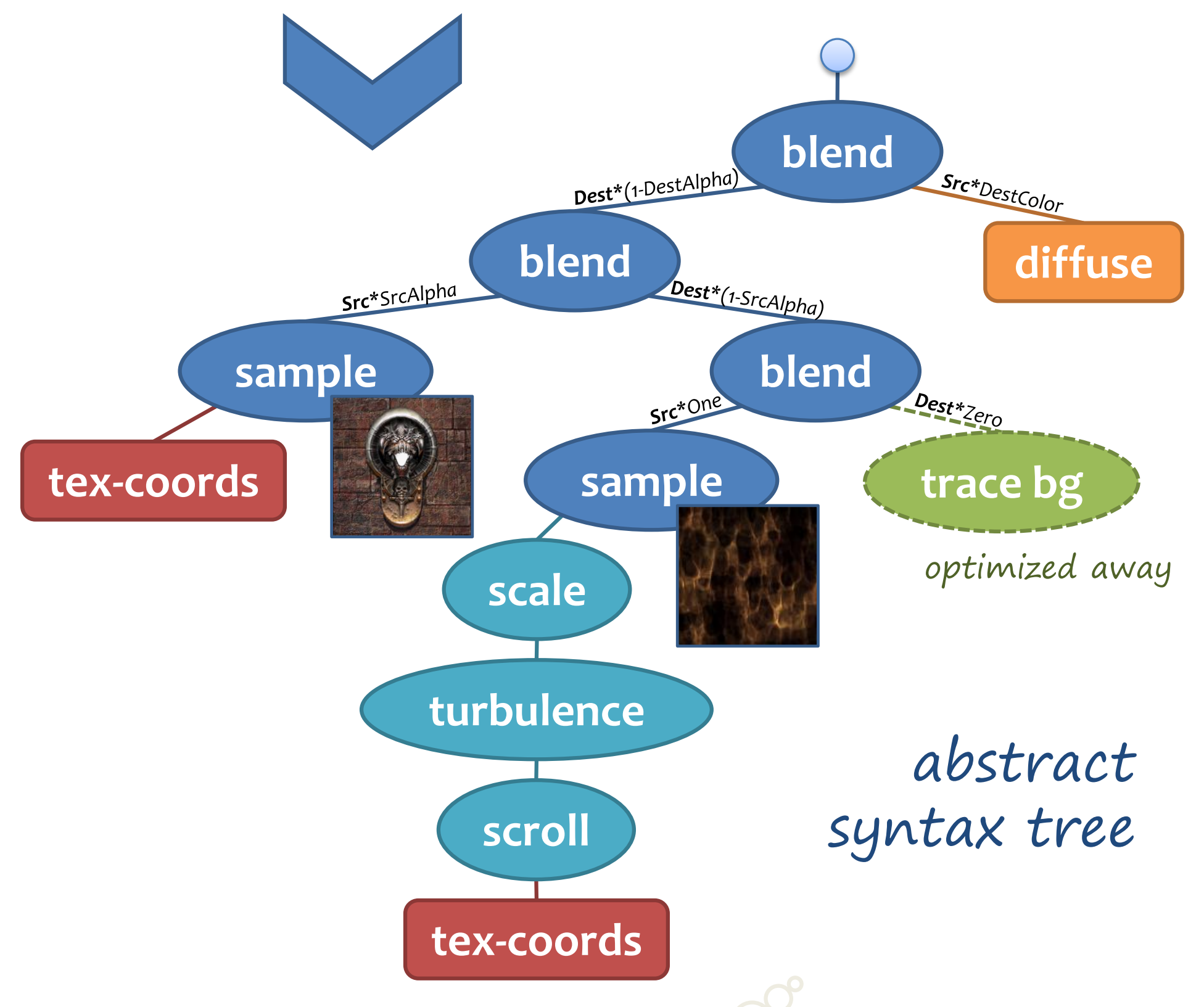
Introduction

id Tech 3 is a game engine used in several well received products, such as id Software's *Quake 3 Arena*, and was released under an open source license in 2005. Within the scope of my diploma thesis I investigated the use of real-time ray tracing in games and found *id Tech 3* to be an excellent basis for evaluating the qualities of ray tracing in creating visually pleasing virtual environments.

id Tech 3 uses a complex material system that allows the specification of the look and the behavior of surfaces in scripts. Support for this system in the new ray tracing based rendering backend was crucial to recreating and augmenting the original look. Noting that interpretation would have resulted in an unacceptable loss of speed, run-time generation of machine code was employed.

Materials in *id Tech 3* can be composed of multiple layers that are blended to get the final color value:
• An animated or static texture can be sampled in each layer at coordinates computed by a chain of modifiers, which apply transformations to the texture coordinates of the surface the material is applied to.
• Color and alpha values can be generated per layer (e.g., based on noise and waveforms) and may be used to modulate the texture color (typically used for lighting).
• Alpha testing, fog, environment mapping, …

```
textures/gothic_block/demon_block15fx
{
  {
    map textures/sfx/firegorre.tga
    tcmod scroll 0 1
    tcMod turb 0 .25 0 1.6
    tcmod scale 4 4
    blendFunc GL_ONE GL_ZERO
  }
  {
    map textures/gothic_block/demon_block15fx.tga
    blendFunc GL_SRC_ALPHA GL_ONE_MINUS_SRC_ALPHA
  }
  {
    map $lightmap
    blendFunc GL_DST_COLOR GL_ONE_MINUS_DST_ALPHA
  }
}
```



```
Value *EmitAdd(Value *a, Value *b) {
  return Builder()->CreateAdd(a, b);
}
```

Code generation using the *Low-Level Virtual Machine*

The LLVM's code generation interface is object-oriented and built on the notion of value-objects, which are created directly by wrapping constants or indirectly as the result of emitting virtual instructions (e.g., `alloca`, `load`, `add`, `cmp`, `call`, ...) with other values supplied as input. Code is arranged in basic blocks, which are linked with branches to implement control flow in a function.

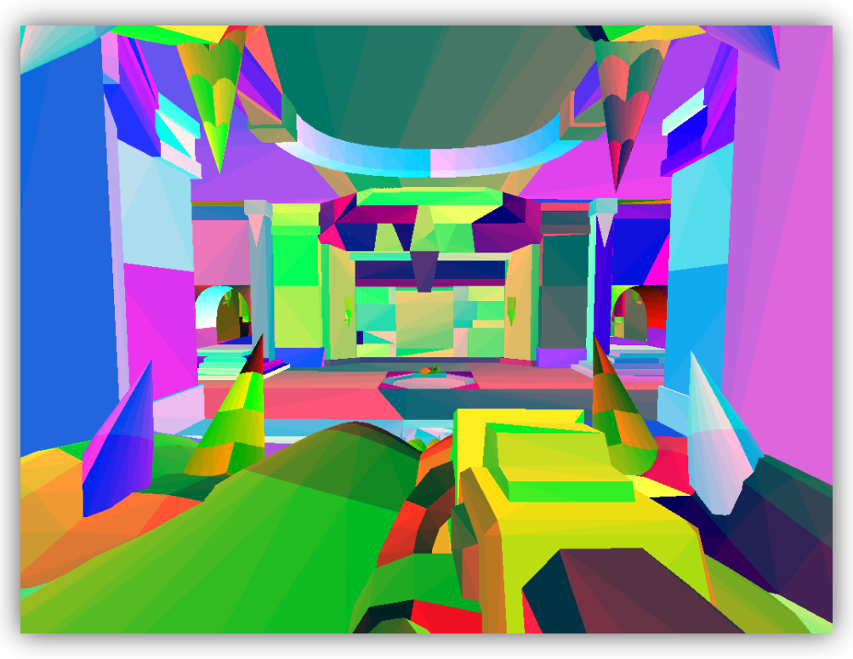
Code generation using the LLVM is type-safe and supports the composition of complex data structures from primitives, such as `ints` and `floats`. Vectors are also natively supported for the emission of SIMD-style code.

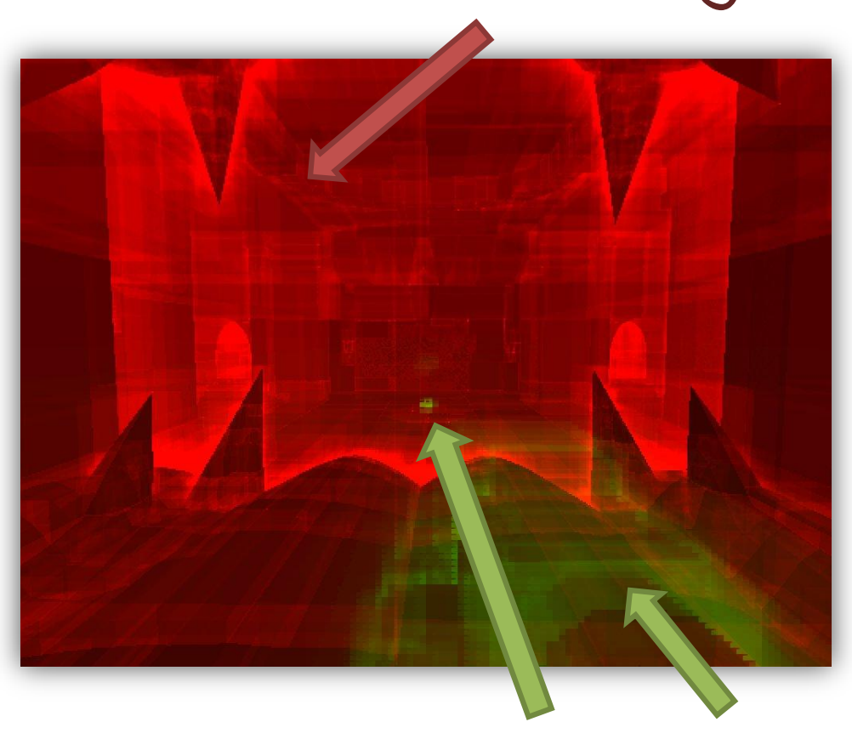
The basis for code generation is an abstract syntax tree derived from the parsed material script. It describes the operations involved in calculating the surface color based on inputs such as texture coordinates, normals and lighting information. Code is emitted as the tree is traversed in postorder with nodes returning values (*a.k.a.* items) that record the result of the represented operation.

After the body of a function has been created a variety of optimizations can be applied to the code. In order to be able to invoke the function it has to be passed to the jitter, which returns a pointer to the generated machine code.

A ray tracing based rendering backend for *id Tech 3*

The rendering backend of the *id Tech 3* engine serves as a connection between the graphics API and the engine's front end, which is responsible for maintaining the state of the virtual environment. In each frame the backend receives a stream of rendering requests (e.g., to draw an object or a list of triangles), applies filtering, such as frustum culling, and issues the corresponding draw calls. In order to supply a ray tracer with a complete description of the environment, a few modifications had to be introduced:
• Dynamic geometry (players, items, …) is collected each frame by intercepting draw calls within the engine and is stored in a BVH. Frustum culling is not applied and the PVS is ignored.
• Static geometry (i.e., walls but not doors) is extracted and stored in a kd-tree when the scene is loaded.

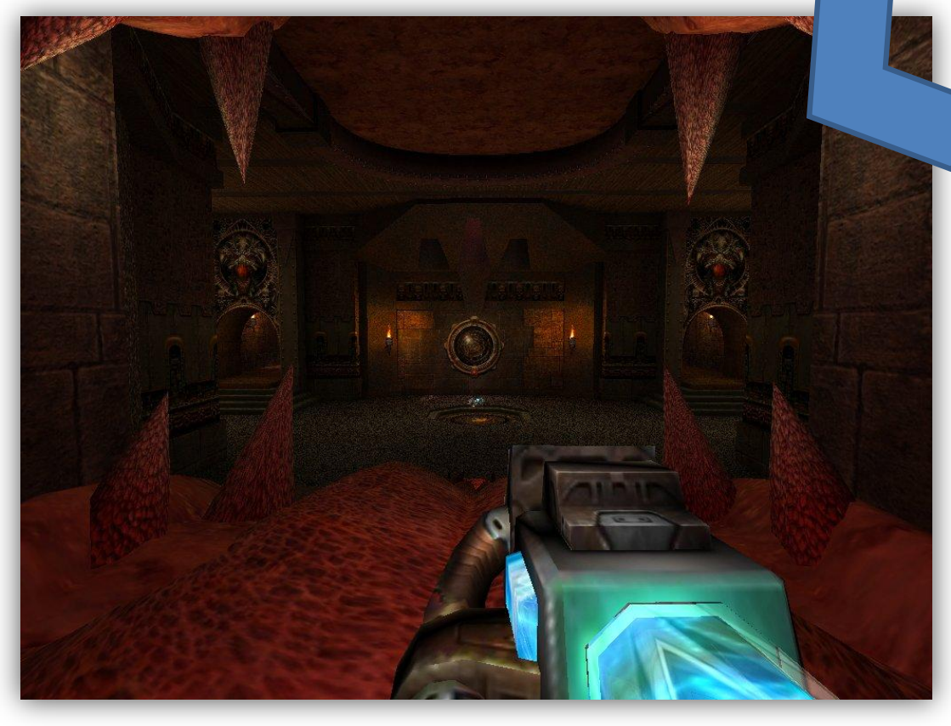


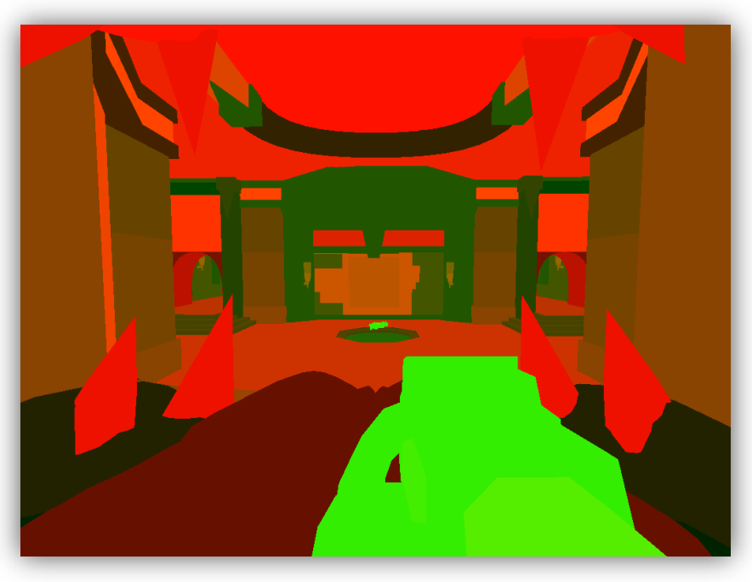


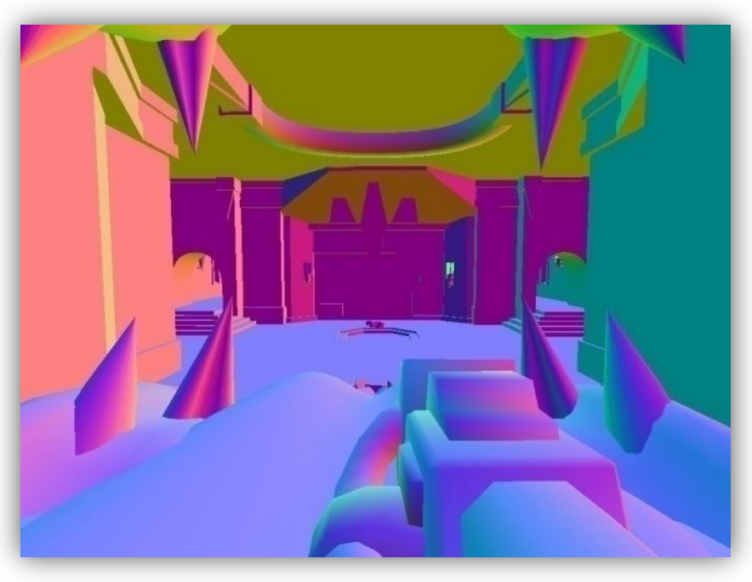
Creation of materials for the ray tracer is implemented by extending the engine's internal script parsing function to load the referenced textures and to generate the shading code using the *LLVM*.

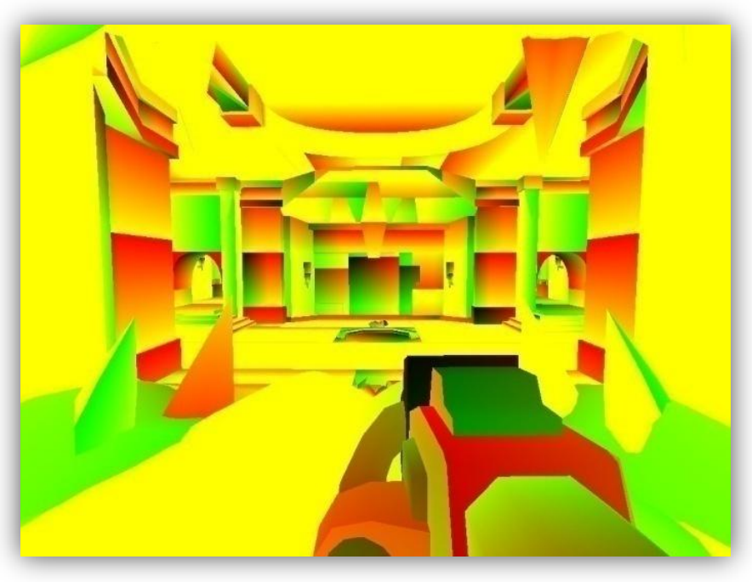
Each frame is rendered with ray tracing after the dynamic geometry has been collected. Rays are spawned for tiles of 8x8 pixels and traced by first traversing the kd-tree followed by the BVH. Shading is performed on groups of rays that hit the same primitive:
• Depending on the requirements of the associated material, normals, texture coordinates, and lighting information (dynamic lights & static lighting in the scene's lightmap) are gathered.
• The shading function is then invoked to determine the color of the hit points. Texture sampling or tracing of secondary rays (e.g., for reflections) may be requested by the function in the process.

In order to facilitate comparisons of the original and the new ray tracing based look of the game, toggling of rendering modes is supported at run-time. Furthermore it is possible to display traversal statistics, material-ids, texture coordinates and normals.

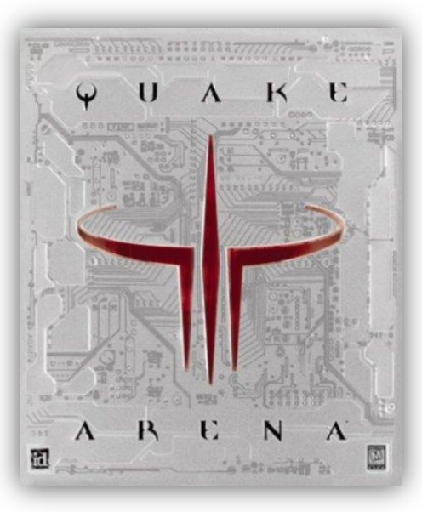




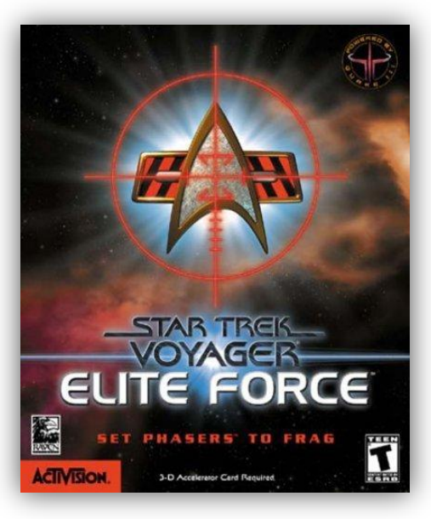




Results

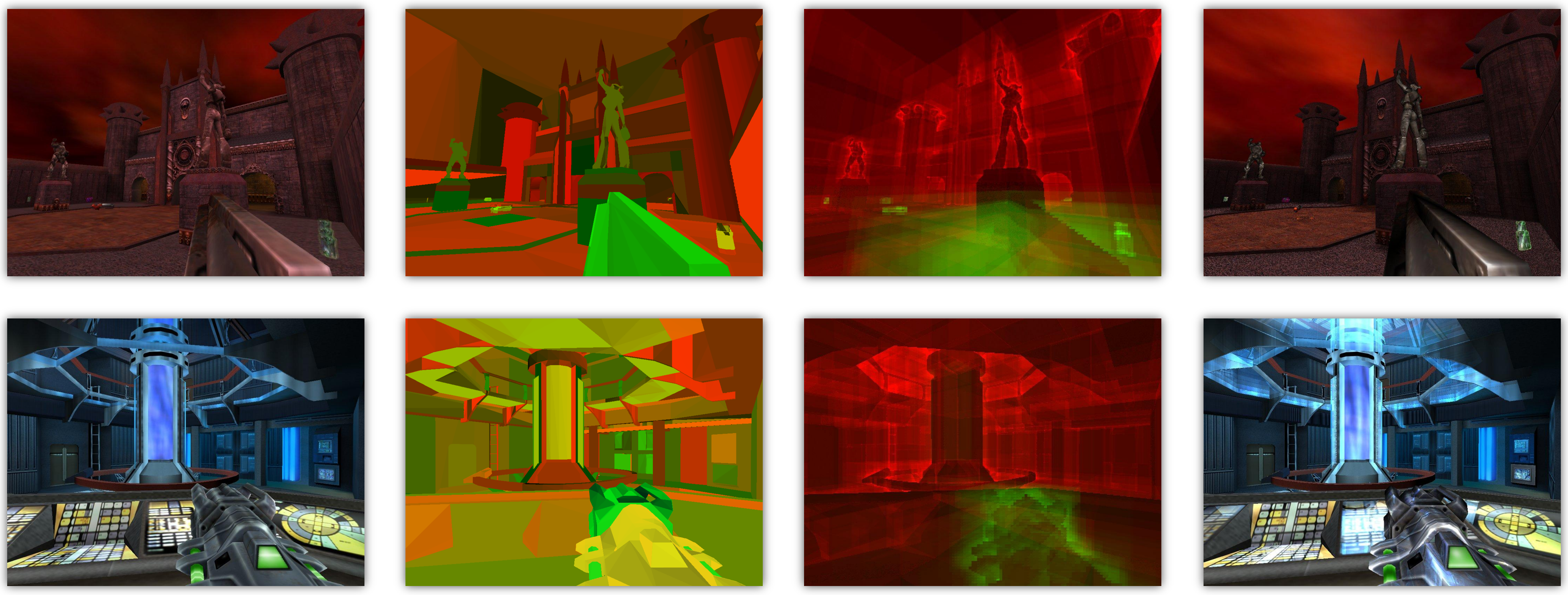


id Software's *Quake 3 Arena*, 1999.
Map: q3dm1, 25k triangles
RT: 6 fps @ 800 x 600



Raven Software's *Star Trek Voyager: Elite Force*, 2000.
Map: Breach, 90k triangles
RT: 1 fps @ 800 x 600

Images rendered on an AMD Athlon X2 3800+ (2005) with 3 GB RAM running 64-bit Ubuntu Linux 7.10.



Original **Triangles** **Traversal** (max. 128 steps) **Ray Traced** (with added reflections)