10

Rendering Optimizations in the Turbulenz Engine

David Galeano

10.1 Introduction

The Turbulenz Engine is a high-performance open source game engine available in JavaScript and TypeScript for building high-quality 2D and 3D games.

In order to extract maximum performance from both JavaScript and WebGL, the Turbulenz Engine needs to reduce waste to the minimum, for the benefit of both the CPU and the GPU. In this chapter, we focus on the rendering loop and the removal of waste originating from redundant and/or useless state changes. In order for this

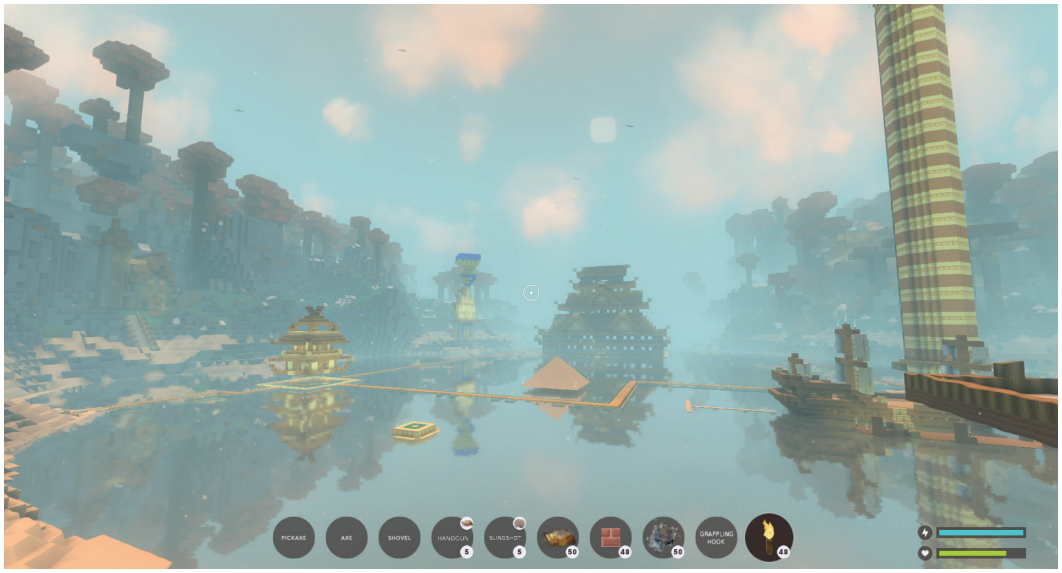

Figure 10.1

The game *Oort Online*.

removal to be optimal, the engine employs several strategies at different levels, from high-level visibility culling, with grouping and sorting of geometries, to low-level state change filtering.

Throughout the chapter we give specific examples from our game *Oort Online* (Figure 10.1), a massive multiplayer game set in a sandbox universe of connected voxel worlds.

10.2 Waste

JavaScript is now a high-performance language. It is still not as fast as some statically typed languages, but fast enough for using WebGL efficiently in order to achieve high-quality 3D graphics at interactive frame rates. Both JavaScript and WebGL are perfectly capable of saturating a GPU with work to do. See [Echterhoff 14] for an analysis of WebGL and JavaScript performance compared to a native implementation.

However, saturating a CPU or GPU with work that is either redundant or useless would be a waste of resources, and identifying and removing that waste is our focus.

There are several kinds of waste that we classify into:

1. Useless work: doing something that has no visible effect
2. Repetitive work: doing the expensive state changes more than once per frame
3. Redundant work: doing exactly the same thing more than once in a row

We will tackle each one in turn, but first let's discuss the cost of removing unneeded work.

10.3 Waste Avoidance

Waste avoidance is simply avoiding the production of waste. It works on the principle that the greatest gains result from actions that remove or reduce resource utilization but deliver the same outcome.

There is always a price to pay for removing waste. In order to save CPU and GPU time, we first need to use some CPU time, and there will always be a trade-off. Too much CPU time spent on filtering out useless work may actually make our application run slower than if there was no filtering at all, but the opposite is usually also true.

The Turbulenz Engine is data-oriented and mostly data-driven; some special rendering effects may be handwritten to call the low-level rendering API directly for performance reasons, but the bulk of the work is defined at runtime based on loaded data.

In general, it is much cheaper to avoid waste at a high level than at a lower level. The higher level has more context information to use and can detect redundancy at a higher scale. The lower levels can only deal with what is known at that instant. For example, the scene hierarchy and the spatial maps available at the high level together allow culling of nonvisible objects in groups, instead of having to check visibility individually for all objects.

The Turbulenz Engine avoids waste at different levels, each one trying to reduce work as much as possible for the lower ones. We will explain the different strategies used at every level.

10.4 High-Level Filtering

This is where the scene is managed, passing information to the middle-level renderer.

At this level, we mostly focus on trying to remove work that provides no usable result. Examples of this kind of waste are geometries that are not visible because they are

- Out of the view frustum
- Fully occluded by other geometries
- Too far away and/or are too small for the target resolution
- Fully transparent

This list also applies to other game elements like, for example, lights affecting the scene, animation of skinned geometries, entities AI, 3D sound emitters, etc.

To optimize away each of these cases, we require extra information that is only available at a high level, for example:

- View frustum
- Geometry AABB
- Dimensions of the rendering target
- Transparency information

The Turbulenz Engine filters out elements that are not visible using a two-step frustum culling system. First we cull the scene nodes and then their contents. Our scene hierarchy has an AABB on each node that contains renderable geometry or lights. Each scene node

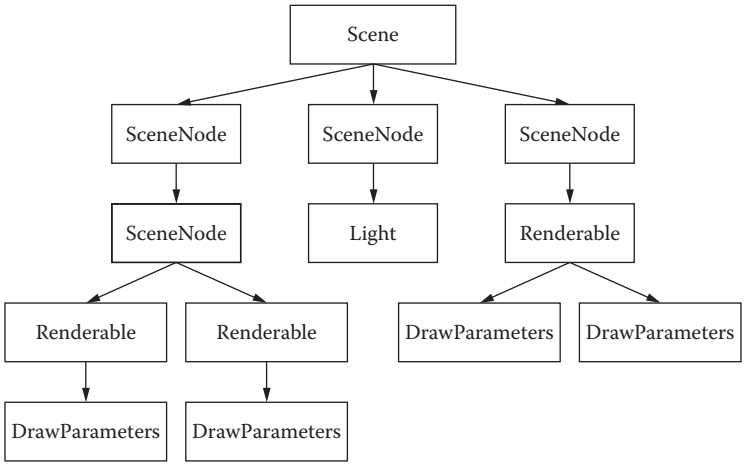

Figure 10.2

Scene example.

can contain an unlimited number of lights or renderable geometries. Figure 10.2 shows an example of a scene. These scene node AABBs are added to spatial maps, one for static objects and a separate one for dynamic objects. They are separate because they are updated with different frequency and we can use different kinds of spatial maps for each case. The spatial maps supported out of the box are

- AABB trees with different top-down building heuristics for static or dynamic objects
- Dense grids
- Sparse grids

For example, our game *Oort Online* uses AABB trees for static objects and dense grids for dynamic ones. Figure 10.3 shows a screen capture showing the renderables bounding boxes.

Once a node is deemed visible then we proceed to check for the visibility of each of the renderables and lights that it contains. When there is a single object on the node, then there is no need to check anything else and it is added to the visible set. When there are multiple objects, we check whether the node was fully visible in the frustum—if so, then all its contained objects are also visible; otherwise, we check visibility for each of them separately. For lights, we may go an extra step by projecting the bounding box into the screen to calculate how many pixels it would actually light, discarding the light or disabling its shadows' maps depending on its contribution to the scene. However, all this work only removes objects outside the view frustum; it does not do anything about occluded objects.

For occlusion culling, the main trade-off is that doing occlusion culling on the CPU is usually very expensive and hence only applied in a limited number of cases. Our game *Oort Online* is made of cubes and hence it is much simpler to calculate occluder frustums

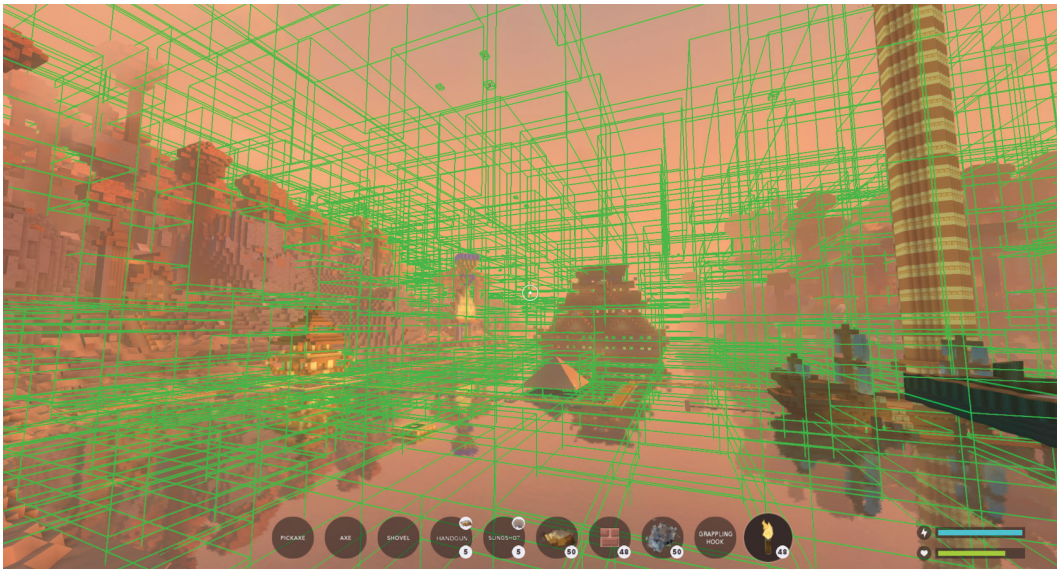

Figure 10.3

Bounding boxes of visible renderables.

than in games using complex nonconvex geometry, but even in this case we only apply occlusion from chunks of 16 × 16 × 16 blocks of opaque cubes closer to the camera. Unfortunately, it only helps in a limited number of situations—for example, looking into a mountain from its base where most of the voxels inside the mountain will be discarded. In other games we employed portals to find the potentially visible set of elements from a particular camera viewpoint.

Once we have removed as much useless work as possible without spending too much time on it, then it is time to pass that information to the next level down. The output from this level is a collection of arrays of visible objects: visible nodes, visible renderables, visible lights, visible entities, etc. We are going to focus just on the list of renderables.

10.5 Middle-Level Rendering Structures

Each visible renderable can represent several rendering geometries. Each rendering geometry is defined by a `DrawParameters` object. Managing the collection of visible rendering geometries for optimal dispatching is the role of the next level down.

`DrawParameters` contains all the information required to render a particular geometry with a particular shading technique:

- Vertex buffers: could be more than one
- Vertex offsets: start of the used region on the vertex buffers
- Vertex semantics: to match a particular vertex component with a particular shader input

- Index buffer: optional, for indexed primitives
- Index buffer offset: start of the used region on the index buffer
- Primitives count: the number of primitives to render
- Primitive type: triangle list, triangle strip, etc.
- Technique: the shading technique to be used to render this geometry
- Technique parameters: a dictionary containing the custom shading parameters for this geometry—for example, the world location, the material color, the diffuse texture, etc.
- User data object: used to group geometries by framebuffer, opacity, etc.
- Sort key: used to sort geometries belonging to the same group

Both user data and sort key are the keys to filtering out waste at this level.

10.5.1 Group Index on the User Data Object

The user data object can be used by the game to store anything about this particular instance of the geometry; the relevant part for this chapter is that the game generally uses this object to tell the renderer what particular group this geometry belongs to.

In hindsight, we should have named this property differently and we should have separated out its different components into different values. The group information is stored as an integer index into the array of groups that the renderer manages; this is game- and renderer-specific.

For example, these are the groups used for our game *Oort Online*:

- Prepass: for smoothing the normals in screen space on geometry close to the camera
- Opaque: for opaque voxels, entities, and opaque vegetation like tree trunks
- Alpha cutouts (or decals): for grass, leaves, etc.
- Transparent: for transparent voxels or entities
- Effects: for particle systems like the smoke or flames from the torches
- Water
- Lava
- Clouds

The different groups for nonopaque geometry that require alpha blending (water, lava, transparent, clouds, etc.) are rendered in different order depending on the camera direction and vertical position in order to get better visual results; otherwise, the different layers may render on top of each other incorrectly.

Other games may have more or fewer groups depending on the renderer. For example, games with dynamic shadows implemented using shadow mapping would have one or more groups for each of the shadow maps.

10.5.2 Sort Key

The `DrawParameters` sort key is a JavaScript number used to sort geometries belonging to the same group. As is often the case in JavaScript, it can be a floating-point value or an integer; it is up to the game renderer to set it correctly. This value can be used to sort the `DrawParameters` objects in ascending or descending order, depending on what makes more sense for its particular render group.

For example, in the game *Oort Online*, the keys are calculated as follows:

- For opaque geometry or for additive-only blending:

```
(distance << 24) |
(techniqueIndex << 14) |
(materialIndex << 8) |
(vbIndex & 255)
```

  The `distance` value is the distance to the camera near plane quantized into 64 logarithmic-scaled buckets.
  The `techniqueIndex` value is the unique shading technique id.
  The `materialIndex` value is the unique material id.
  The `vbIndex` value is the unique vertex buffer id.
- For transparent geometry:

```
(distance * 1024 * 16) | 0
```

  The `distance` value is the distance to the camera near plane in world units.

For opaque geometries, the main reason for sorting is performance. We build the sorting key in such a way that geometries with the same technique, material, or vertex buffer are one after the other in the sorted list. A perfect sorting would require a multilevel bucket hierarchy, which could be too expensive to build, while the sorting key provides a cheap enough solution being fast to generate and fast to sort by. The higher bits of the key are used to sort for the most important state change to optimize by, while the lower bits are left for the cheaper changes. The order will usually depend on the scene complexity. If overdraw is a significant issue, then we will use the distance to the camera on the higher bits, while in some other cases it is the shading technique that requires optimization, either because overdraw is not significant or because the hardware can deal with it efficiently—for example, on tile-based deferred GPUs.

For transparent geometries, the main reason is correctness. Transparent geometries are sorted by distance to get the correct rendering, from back to front. However, if the blending is additive-only, then this is not needed. In the case of our game *Oort Online*, the "effects" pass only contains geometries rendered with additive blending, which is correct no matter the order; this means that we can employ more efficient sorting for this group. Unfortunately many rendering effects do not fall into this category; in our game *Oort Online*, only a handful of particle systems use additive blending—for example, the sparks coming out of a torch.

In general, the Turbulenz Engine tries to use integers smaller than 30 bits in order to force the JavaScript JIT compilers into using integer operations for sorting instead of floating-point ones. Comparing integers is significantly cheaper than comparing floating-point values. The JIT compiler will optimize for either integers or floating-point values, but performance is more predictable if we use the same types everywhere, every time. For more information about low-level JavaScript optimizations, see [Wilson 12].

10.6 Middle-Level Filtering

This is where the renderer groups and sorts `DrawParameters` to be passed down to the low-level rendering functions.

This level focuses on removing repetitive work that is done multiple times but not in a row. Examples of this kind of waste include:

- Bind the same framebuffer more than once.
- Enable blending more than once.

To remove this waste, we rely on the game and the renderer to provide enough information to group and sort `DrawParameters` in the most efficient way to reduce the number of state changes to the minimum.

At this point it is worth explaining the real cost of making WebGL calls.

10.6.1 WebGL Costs

Making a function call is never totally free, but there are some functions that are far more expensive than others. Sometimes the real cost of function calls is not immediately obvious; they may have big performance side effects that only appear in combination with other function calls, and most WebGL calls are in this category. Older versions of rendering APIs like D3D9 or OpenGL ES 2 (which WebGL represents) do not actually represent how modern GPUs work. A lot of state and command translation and validation is lazily evaluated behind the scenes. Changing a state may imply just storing the new value and setting a dirty flag that is only checked and processed when we issue a draw call.

This lazy evaluation cost heavily depends on the hardware and the different software layers that drive it, and there are big differences between them; for example, changing frequently from opaque rendering to transparent rendering and vice versa by changing the blend function is very expensive on deferred tile rendering architectures because they force flushing the rendering pipeline for each change, which could be very wasteful if there is a significant amount of overdraw. Defining the specific costs of each change for each piece of hardware is complicated, but there are some general rules that should work fine in most hardware.

The following are in order of cost of change from high to low:

- Framebuffer
- Blend state
- Shader program
- Depth state
- Other render states
- Texture
- Vertex buffer
- Index buffer
- Shader parameters (uniforms)

This ordering may change significantly from hardware to hardware, but as generic rules they are good enough for a start. For each particular platform we should profile and adapt. For more in-depth information, see [Forsyth 08].

Even if the geometries are rendered in the right order to minimize state changes, there is still one big source of GPU overhead that we are also trying to get rid of at this level: overdraw.

10.6.2 Overdraw

Overdraw is one of the main sources of wasted effort on complex 3D environments; classified as useless work, some pixels are rendered but are never visible because other pixels are rendered on top of them. This is a massive waste in terms of data moving through the app and the graphics pipeline.

In theory this kind of waste is cheaper to be removed early on at a higher level; we should detect occluders and remove all occludes from the rendering list. However, this is a lot of work to do on the CPU and most occludees are only half covered, which means that we should, in theory, split the triangles into visible and not visible parts. And if we go down that route, then most of our CPU would be wasted on occlusion calculations while the GPU sits idle waiting for work to do.

Instead, we rely on grouping and sorting opaque geometry based on the distance to the camera near plane. By first rendering opaque geometry closer to the camera, we can use the depth test to discard shading for occluded pixels. It does not eliminate all the waste from vertex transformations and triangle rasterization, but at least it removes the cost of shading in a relatively cheap way for the CPU. This is one case where we optimize for the CPU, relying on the massive parallel performance of modern GPUs.

This optimization makes sense because of early-z reject hardware implementations that provide fast early-out paths for fragments that are occluded; some hardware even employs hierarchical depth buffers that are able to cull several occluded fragments at the same time. This optimization makes less sense on tile-based deferred architectures that only shade the visible fragment when required at the end of the rendering.

If the game is not CPU limited by the number of draw calls, then a z-only pass could be a good solution to remove most of the overdraw waste. Rendering only to the depth buffer can often be optimized to be significantly faster, but an excessive amount of draw calls is too often the norm for complex games, so this solution is not usually more optimal than just trying to sort opaque geometry front to back. A z-only pass is even more handicapped in JavaScript/WebGL because of the additional overhead compared to native code. And, as before, on tile-based deferred architectures, this solution may actually go against the optimizations done in hardware and result in more GPU overhead.

10.6.3 Benefits of Grouping

There are several reasons why we group `DrawParameters` into different buckets:

1. Because they are rendered into different framebuffers
   - For example, they gather screen depth or view space-per-pixel normal information into textures to be used by subsequent groups.
2. Because they need to be rendered in a specific order for correctness
   - For example, transparent geometry should be rendered after all the opaque geometry; otherwise, they may be blending on top of the wrong pixel.
3. Because they need to be rendered in a specific order for performance
   - For example, opaque geometry closer to the camera could be rendered in its own group before the distant ones.
   - For example, enabling and disabling blending may be quite expensive on some hardware, so we try to do it only once per frame.

Points 2 and 3 require clarification. There is some overlap between the sort key and the group index. In theory we could encode all the required information to sort `DrawParameters` for performance reasons into the sort key, but there is a performance limit on the key size and it is relatively easy to use more than 31 bits and then sorting starts to be expensive for big collections. By moving part of the key information into the group index you can increase the effective key size without penalties; this could be seen as some kind of high-level bucket sorting.

Once all our `DrawParameters` are grouped and sorted efficiently, then they are passed to the low-level API for rendering.

For more on draw calls ordering, see [Ericson 08].

10.7 Low-Level Filtering

This is where we dispatch the `DrawParameters` changes to WebGL. Big arrays of `DrawParameters` objects are passed down to the low-level rendering functions for dispatching.

This level focuses on removing redundant work that is done multiple times in a row. Examples of this kind of waste include:

- Binding the same vertex buffer repeatedly
- Binding the same texture on the same slot repeatedly
- Binding the same shading technique repeatedly

To avoid this redundant work, we shadow all the internal WebGL state, including uniforms for shader programs, avoiding changes that do not alter it. There is overhead for keeping this shadow of the WebGL state in both memory usage and CPU cost, but as the rendering has been sorted already to keep redundant changes close together, most of the time the check for superfluous work saves a lot of work due to the high overhead of the underlying WebGL implementations.

To reduce the cost of state checks as much as possible, we need to store the rendering data in optimal ways. The main rendering structure at this level is the Technique.

10.8 The Technique Object

A Technique object contains the required information for shading a given geometry:

- Vertex and fragment shaders
  - A unique program is linked for each combination and is cached and shared between techniques.
- Render states
  - Contains only the delta from our predefined default render states:
    - `DepthTestEnable: true`
    - `DepthFunc: LEQUAL`
    - `DepthMask: true`
    - `BlendEnable: false`

    - `BlendFunc: SRC_ALPHA, ONE_MINUS_SRC_ALPHA`
    - `CullFaceEnable: true`
    - `CullFace: BACK`
    - `FrontFace: CCW`
    - `ColorMask: 0xffffffff`
    - `StencilTestEnable: false`
    - `StencilFunc: ALWAYS, 0, 0xffffffff`
    - `StencilOp: KEEP, KEEP, KEEP`
    - `PolygonOffsetFillEnable: false`
    - `PolygonOffset: 0, 0`
    - `LineWidth: 1`
- Samplers
  - The texture samplers that the shaders would require.
  - They are matched by name to textures from the technique parameters objects passed in by the `DrawParameters` object.
  - The sampler object contains only the delta from our predefined sampling states:
    - `MinFilter: LINEAR_MIPMAP_LINEAR`
    - `MagFilter: LINEAR`
    - `WrapS: REPEAT`
    - `WrapT: REPEAT`
    - `MaxAnisotropy: 1`
- Semantics
  - The vertex inputs that the vertex shader would require.
  - These are some of our predefined semantics:
    - `POSITION`
    - `NORMAL`
    - `BLENDWEIGHT`
    - `TEXCOORD0`
- Uniforms
  - The uniform inputs that the program would require
  - Matched by name to the values from the technique parameters object passed on by the `DrawParameters` object.

The `Technique` objects are immutable; if we want to use the same program with a different render state, we need to create a new `Technique`. This could potentially scale badly, but in practice none of our games use more than a couple dozen techniques. The main reason to have potentially too many techniques would be the need to support too many toggleable rendering configurations—for example, water with or without reflections—but in that case we can just load the techniques that we actually need.

The `Technique` object contains lots of information; when we change a shading technique, we are actually potentially changing many WebGL states, which is why sorting by technique is so important in many cases, although we still need to optimize it as much as possible.

10.9 Dispatching a Technique

When we have to dispatch a new Technique, we still need to reduce the state changes to the minimum. We keep track of the previously dispatched Technique and we only update the delta between the two.

- Program: only changed if different from the previous one
- Render states: the new state values are only applied if they are different from the values previously set. The render states set by the previous technique that are not present on the current technique are reset to their default values.
- Samplers: the new sampling values are only applied if they are different from the values previously set. The sampling states set by the previous technique that are not present on the current technique are reset to their default values.

Once the main states have been updated, the remaining changes are the buffers and the uniforms.

10.10 Dispatching Buffers

Vertex and index buffers are only dispatched if they differ from the current value set by the previous `DrawParameters` object. The Turbulenz Engine creates big buffers that are shared among different geometries, which helps to reduce the number of buffer changes.

Index buffers are easy; if the current buffer is different from the last one, we change it. If the `DrawParameters` does not contain an index buffer because it is rendering an unindexed primitive, then we just keep the old buffer active because it may be needed again in the future.

Vertex buffers are more complicated. For each vertex buffer, we match each of its vertex components to the relevant vertex shader semantic and then we check if, for that semantic, we already have used the same vertex buffer. If so, we do nothing; otherwise, we need to update the vertex attribute pointer for that semantic.

Shader semantics are hardcoded to match specific vertex attributes; for example, `POSITION` is always on the attribute zero. This makes dispatching simpler and faster because the same semantic will always be set to the same attribute, which means that in many cases we do not need to set a vertex attribute pointer again when rendering the same vertex buffer with multiple techniques. When the program is linked, we remap its attribute inputs to match our semantics table. This has potential issues when the vertex shader only supports a very limited number of attributes (which happens sometimes on mobile phones) and it means that the mapping between semantics and attributes is calculated right after creating the WebGL context; however, once it is done it never changes.

If vertex array objects (VAOs) are supported, then we build one for each combination of vertex buffers and index buffer present in the `DrawParameters` objects. As we share the buffers between many different geometries, the actual number of combinations is usually quite low. This allows us at dispatch time to simplify all the buffer checks to a single equality comparison between the current VAO and the previous one. Even when the VAOs are different, setting them with WebGL is cheaper on the CPU than setting all the different buffers and vertex pointer attributes, which makes them a big win for complex scenes.

10.11 Dispatching Textures

Textures are nontrivial to set in WebGL because of the stateful nature of WebGL. At loading time, we assign a specific texture unit to each sampler used by a shader. These units are shared by all the shaders; the first sampler on each shader will use unit zero, the second sampler on each shader will use unit one, and so on. When we need to set a texture to a sampler, first we need to activate the required texture unit (but only if it was not already the active one), and then we need to bind the texture for the required target (`2D` or `CUBE_MAP`). This requires a bit of juggling trying to avoid changing the active unit and the bound texture too many times when changing techniques. Our sort key includes a material id, which is derived from the collection of textures applied to a particular renderable; this way, we try to keep groups of textures together and to minimize the number of times a texture is bound.

An alternative system would bind every texture to a different unit at loading time and then we would need to tell the shader which texture unit to use for each renderable at dispatch time. However, the maximum number of texture units supported varies heavily between video cards, some of them limited to 32 or less. This means that, if we are using more textures than the limit, then we still need to constantly bind textures to different units; we found this much slower in practice than our current system of hardcoded texture units at loading time.

10.12 Dispatching Uniforms

Dispatching the uniforms required for each renderable is accumulatively the most expensive thing our games do—not because changing a uniform is too expensive, but rather because we have lots of them. Potentially, we could have a single big uniform array for each renderable in order to reduce the number of WebGL calls, but that would mean that any tiny difference in the array would require dispatching the whole array (which is a waste), so we group uniforms by frequency of change. Shading parameters that change infrequently between renderables are set in the same uniform array, while dynamic parameters are separated out into individual uniforms. This organization is quite effective in order to minimize the number of changes but it means that some techniques have dozens of individual parameters.

Values for each uniform are extracted at dispatch time from the technique parameters dictionaries stored in each `DrawParameters` object and then compared against the current values set for that particular uniform on the current active program. Only when the values differ do we update the uniform.

To avoid setting the same values twice, we employ a two-level filtering system. We store the JS array that was last set on the uniform; a quick equality check avoids setting the same object twice in a row for the same uniform. This works quite well in practice because of the sorting by material and because we reuse the same typed arrays for the same kinds of data as much as possible. Of course this information is only valid within the function that dispatches an array of `DrawParameters` and needs to be reset once it finishes. If the uniform consists of a single value, then we skip this level.

Once the JS array is determined to be different from the previous one, then we check each individual value on the array. By default we use equality checks for each value on

each uniform, which seems slow but is vastly faster than actually changing the uniform when it is not required. But, when the uniform contains a single floating-point value, we do not use an equality check; we compare whether the absolute difference between the new value and the old value is greater than 0.000001 and only then do we update it. This could be potentially risky if the precision required for that value is higher than the threshold, but we have never found a problem in practice. We could be more aggressive and apply the threshold update requirement for all uniforms, or even use a lower threshold, but we did find rendering issues when doing so. In particular, some rotations encoded in matrices require quite a lot of precision to be correct. However, this is something that could potentially be enabled per project; many users will not notice that an object did not move the tenth of a millimeter that it should have moved.

10.12.1 Performance Evaluation

A typical frame in the game *Oort Online* would have metrics similar to these:

- 9.60 ms total dispatch time
- 3,316 uniform changes
- 2,074 draw calls
- 68 technique changes
- 670 vertex array object changes
- 3 index buffer changes
- 19 vertex buffer changes
- 23 vertex attribute changes
- 78 render state changes
- 181 texture changes
- 32 framebuffer changes

These numbers were captured on a machine with the following specs:

- NVIDIA GeForce GTX 750 Ti
- Intel Core i5-4690 CPU 3.50 GHz
- Ubuntu 14.10
- Google Chrome version 39

After disabling the low-level checks for equality of values, the metrics change to

- 10.53 ms total dispatch time
- 7,477 uniform changes

Disabling also the higher level checks for JS array equality changes the metrics to

- 11.56 ms total dispatch time
- 16,242 uniform changes

These numbers vary a lot depending on the camera position in this heavily dynamic game, but the gains are consistent in any situation. Each of our filtering levels reduces the number of uniform changes by almost 50% and reduces the total dispatch time by about 10%.

10.13 Resources

The Turbulenz Engine is Open Source and can be found at https://github.com/turbulenz/turbulenz_engine. The documentation for the Turbulenz Engine is online and can be found at http://docs.turbulenz.com/

Bibliography

[Echterhoff 14] Jonas Echterhoff. "Benchmarking unity performance in WebGL." http://blogs.unity3d.com/2014/10/07/benchmarking-unity-performance-in-webgl/, 2014.

[Ericson 08] Christer Ericson. "Order your graphics draw calls around!" http://realtime-collisiondetection.net/blog/?p=86, 2008.

[Forsyth 08] Tom Forsyth. "Renderstate change costs." http://home.comcast.net/~tom_forsyth/blog.wiki.html#[[Renderstate%20change%20costs]], 2008.

[Wilson 12] Chris Wilson. "Performance Tips for JavaScript in V8." http://www.html-5rocks.com/en/tutorials/speed/v8/, 2012.