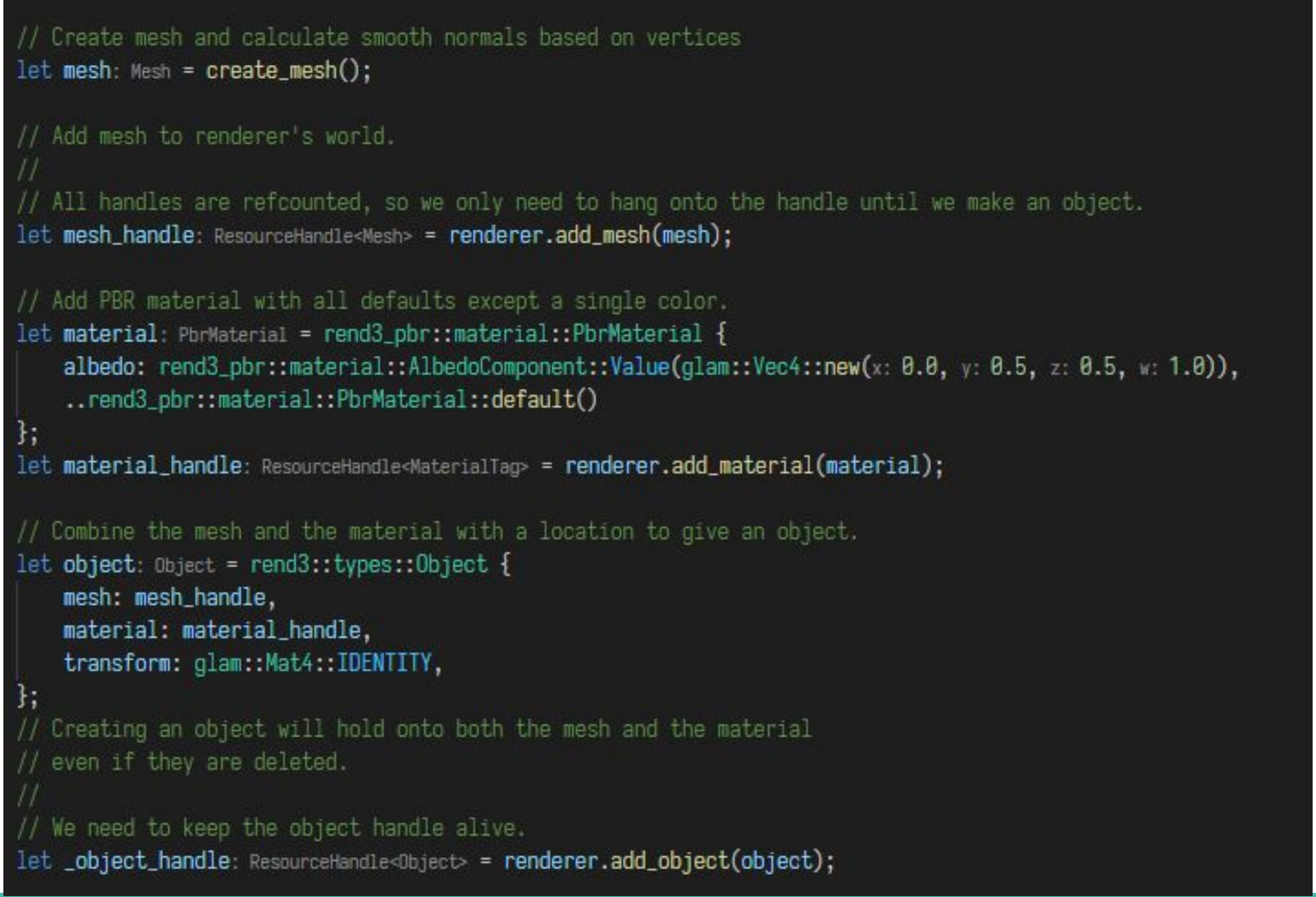
rend3's Architecture: Efficient, Customizable Rendering

cwfitzgerald

What is rend3?

A 3D Rendering Library built on wgpu.

- Create Meshes
- Create Textures
- Create Materials w/ those Textures
- Create Objects w/ (Mesh + Material + Transform)


```rust
// Create mesh and calculate smooth normals based on vertices
let mesh: Mesh = create_mesh();

// Add mesh to renderer's world.
//
// All handles are refcounted, so we only need to hang onto the handle until we make an object.
let mesh_handle: ResourceHandle<Mesh> = renderer.add_mesh(mesh);

// Add PBR material with all defaults except a single color.
let material: PbrMaterial = rend3_pbr::material::PbrMaterial {
    albedo: rend3_pbr::material::AlbedoComponent::Value(glam::Vec4::new(x: 0.0, y: 0.5, z: 0.5, w: 1.0)),
    ..rend3_pbr::material::PbrMaterial::default()
};
let material_handle: ResourceHandle<MaterialTag> = renderer.add_material(material);

// Combine the mesh and the material with a location to give an object.
let object: Object = rend3::types::Object {
    mesh: mesh_handle,
    material: material_handle,
    transform: glam::Mat4::IDENTITY,
};
// Creating an object will hold onto both the mesh and the material
// even if they are deleted.
//
// We need to keep the object handle alive.
let _object_handle: ResourceHandle<Object> = renderer.add_object(object);
```

What is rend3?

3D Rendering Library that is designed to be:

- Easy to Use
- Customizable
- Efficient

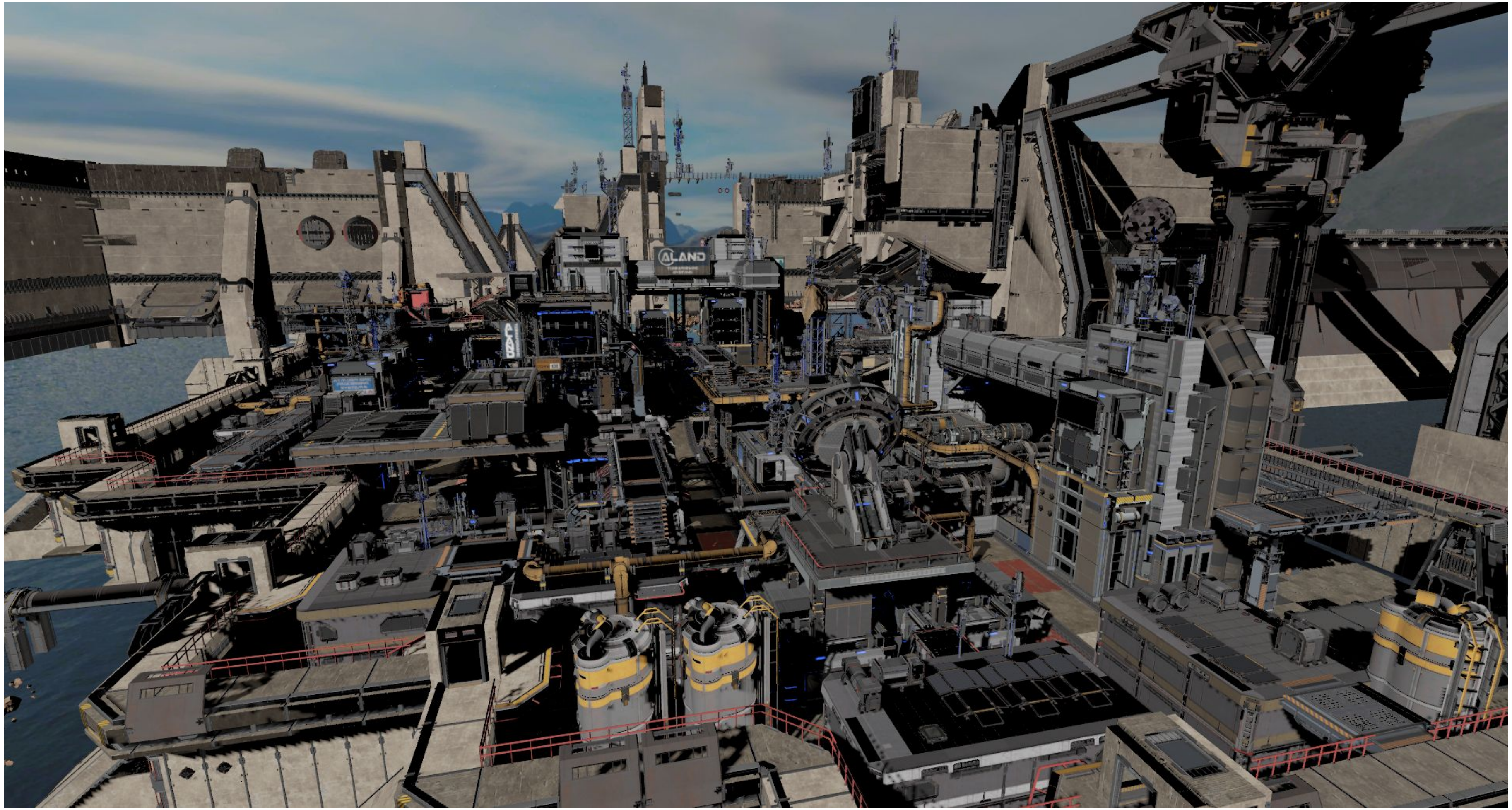
Easy to start with, possible to ship on.

These goals can often be opposing. This talk will be discussing about how to balance these.

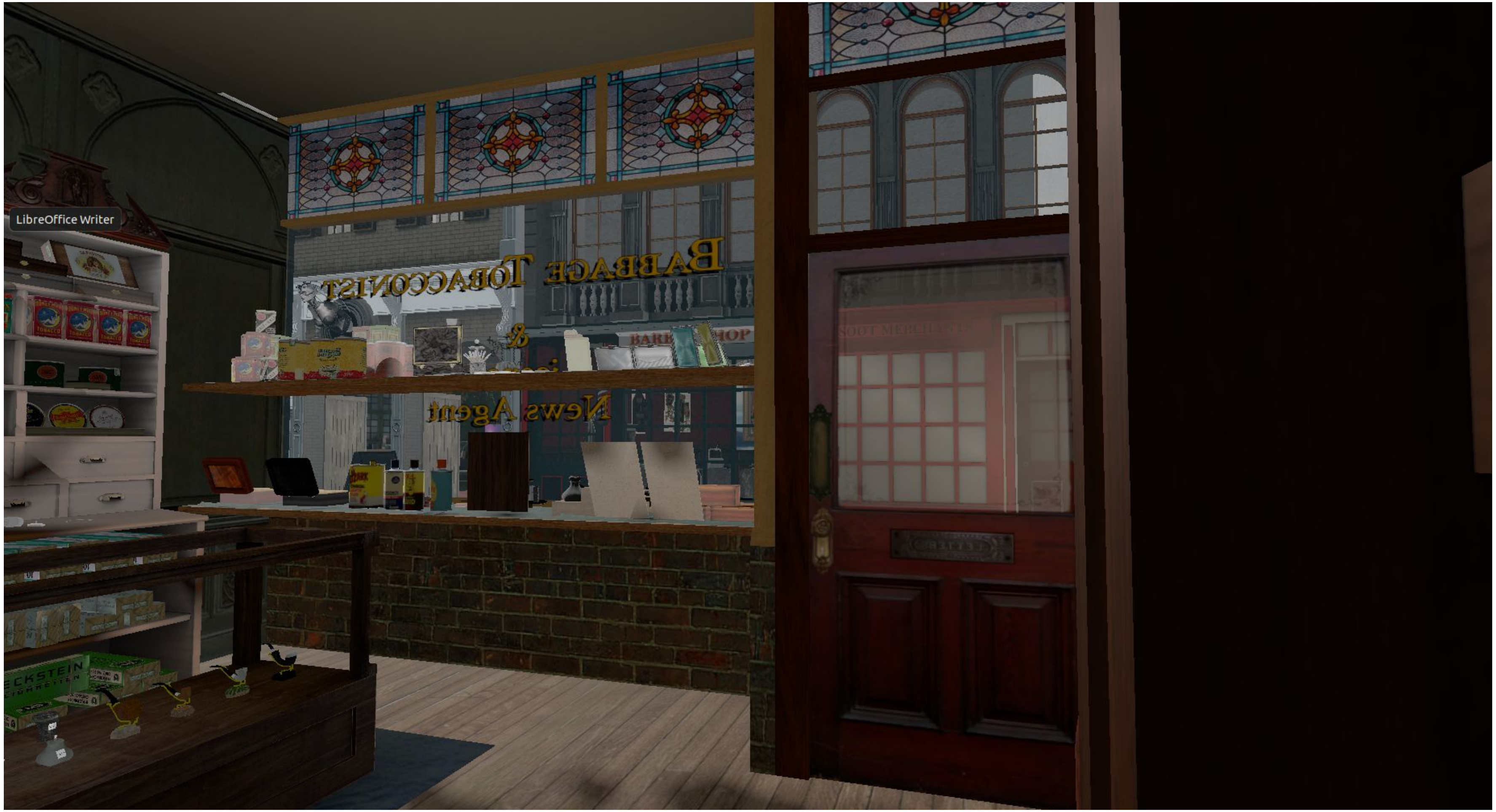
What is rend3?

- "Just the renderer" of an engine.
  - Aimed at people using unity/unreal for the renderer.
- Easy to get started, easy to use.
- As much performance as possible without sacrificing ease.
- Designed for use in real production software, not just toys.
  - Need to support custom graphics of all kinds.
  - Need to scale well, even to large scenes.
  - Stable, predictable performance.

Animats Viewer



scene-viewer example



Animats Viewer

A Talk In Three Parts

The balance between requirements is struck in three ways.

- Careful scope reduction.
- Ubiquitous defaults.
- Optimized use of data structures.

Part 1: Scope Reduction

Scope Reduction

- “3D Rendering library” is not well defined
- GPUs get used for more than just rendering.
    - Object selection
    - Simulation
- If I tried to solve every problem, I would never release rend3….
    - If fact I almost never did.

Dangers of Scope Creep

- Can and has killed projects
- rend3 must to support custom shaders
    - Real games need custom shaders for objects.
    - Tried multiple solutions.
- Each solution was more general than the last
    - Required native support for every situation.
- Took 8 months to find a solution.
- All the momentum rend3 was building died.

The Curse of Renderlists

- Initial solution to the problem is custom shaders was a “renderlist”
- Like a rendergraph, but without the auto-scheduling
- For every single kind of operation a user would have to do, the renderlist must support it.
- This is a crazy amount of stuff to support.
- I didn’t think about what my actual problem was.
- I have a beautiful way of specifying what to do: wgpu itself.
- Why solve an already solved problem because I can.

Solve Problems, don’t Problem Solutions

- When you are designing a solution, SOLVE A PROBLEM!
  - Do not invent problems to solve.
  - “Oh I could also just solve that” is endless scope creep.
- Must know problem parameters in order to design a useful solution.
  - In order to know problem parameters, you must have that problem at all.
- Waiting to solve a hypothetical problem has never sunk a project
- Solving a hypothetical problem has.

Part B: Ubiquitous Defaults

A Solution for Customizability

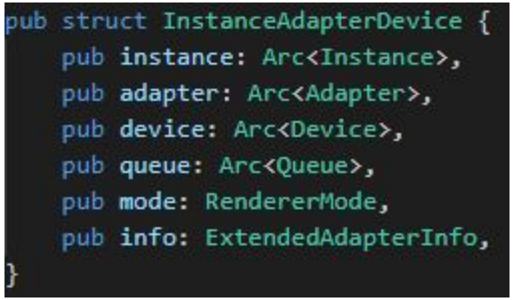
- When doing complex things, there needs to be the ability to customize that behavior.
- Tons of options are extremely intimidating when making simple projects or for users starting out.
- A lot of situations, all of those options aren’t needed.

Defaults

- A useful solution to that is by creating defaults for those options.
- Model common or simple solutions and provide those.
- By having helpers, you move the code “into user space” which makes it really easy to replace.

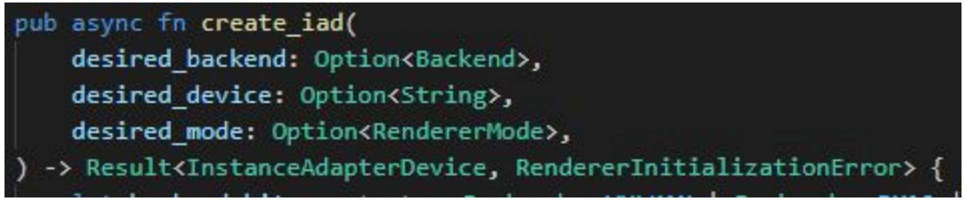
Simple Example

- Instance/adapter/device creation is nuanced.
- User shouldn’t need to write the boilerplate to create the IAD.
    - Not *hard* but a hundred so lines of code to enumerate adapters and choose an ideal one.
    - Even simple creation takes 40-50 LOC.
- Must be able to use user created IAD.
    - Fitting into an existing program needs to use an already made device.



```rust
pub struct InstanceAdapterDevice {
    pub instance: Arc<Instance>,
    pub adapter: Arc<Adapter>,
    pub device: Arc<Device>,
    pub queue: Arc<Queue>,
    pub mode: RendererMode,
    pub info: ExtendedAdapterInfo,
}
```

Flexible



```rust
pub async fn create_iad(
    desired_backend: Option<Backend>,
    desired_device: Option<String>,
    desired_mode: Option<RendererMode>,
) -> Result<InstanceAdapterDevice, RendererInitializationError> {
```

Simple

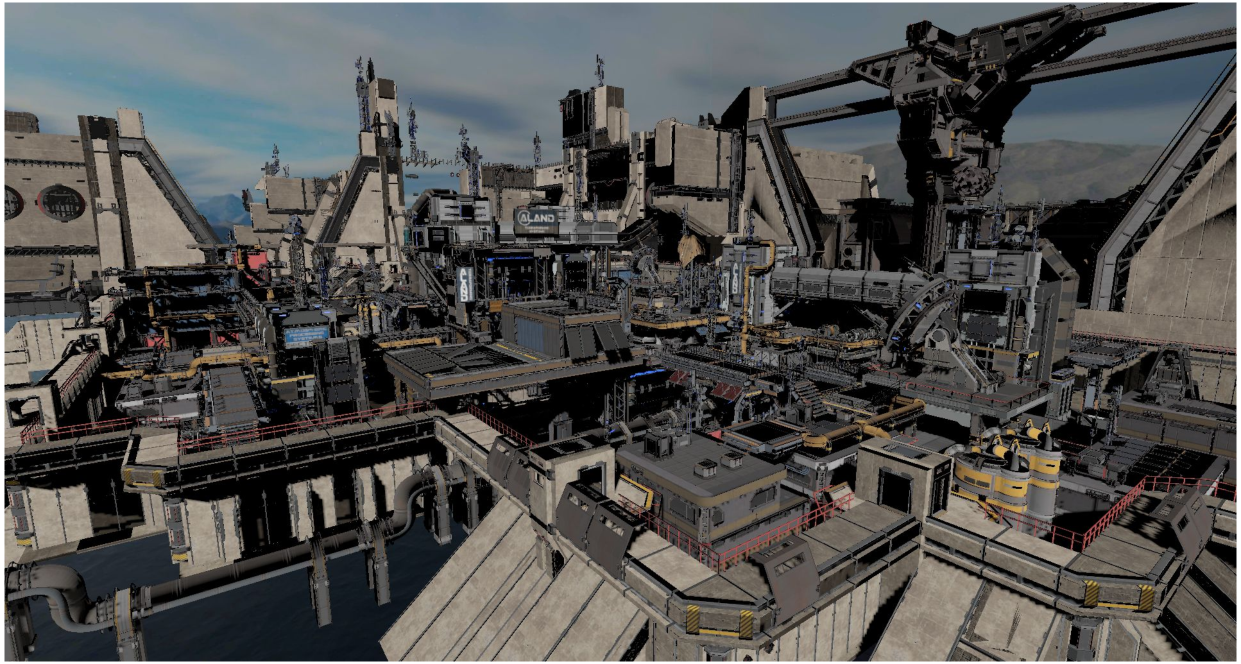
Complex Example

- Render Routine does all the actual rendering.
- Need to be able to compose and implement your own rendering.
- Every user of rend3 shouldn’t need to write the very common situations:
  - Unlit Shaders
  - PBR Shaders
- Composed of small, re-usable parts.
  - When someone needs to roll their own, they can take those existing parts.

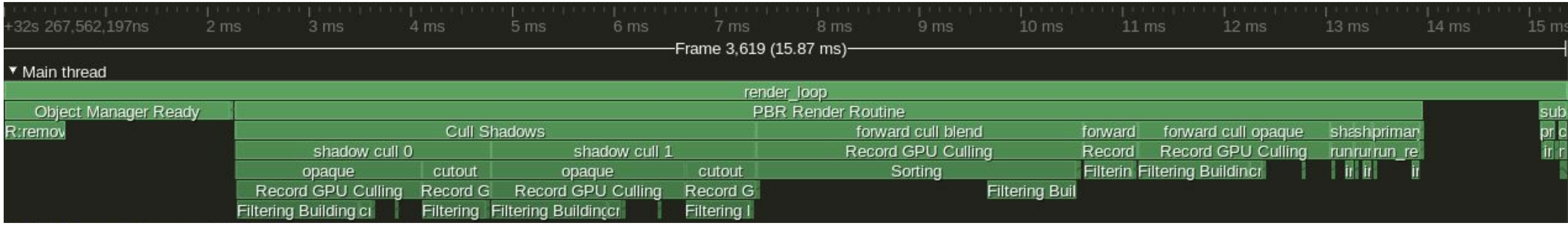
Part III: Performance

Performance

50,000 objects, 20M verts. Rendered to 4 cameras.
16ms CPU on a 10850k @ 5GHz, ~15ms GPU on a 1080ti.



Performance Requirements

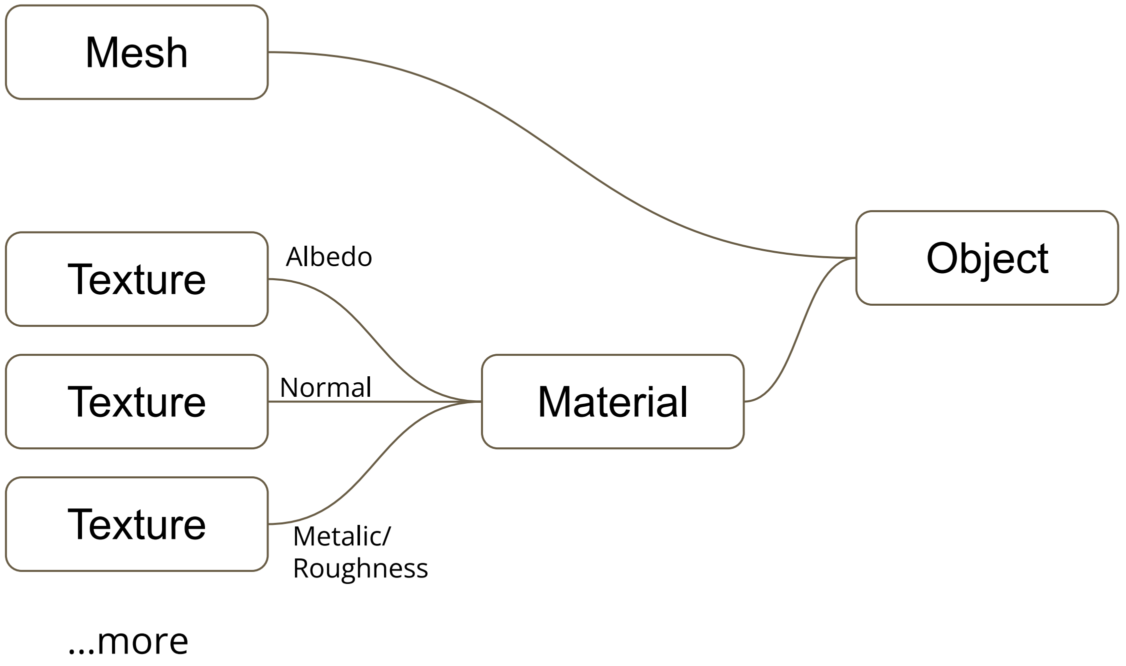


15.5ms CPU loop.

We need to be able to run at more than 60fps for this scene.

Improving GPU performance is possible, but involved, let's get this CPU side down first.

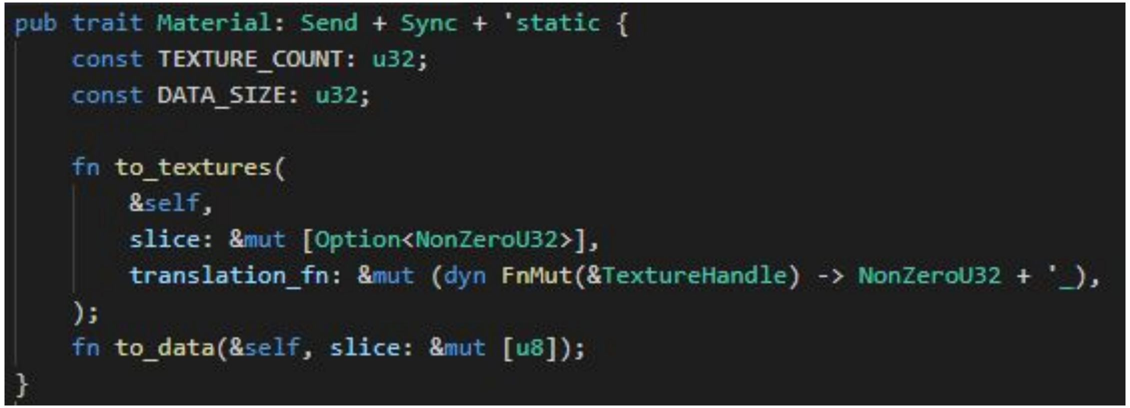
An Overview of rend3's Object System



- Materials are Routine specific.
- Need to store materials of arbitrary types.
  - We need to define what a material *is*.
- Need to store objects pointing to any type of material.

How Do? The Trait

- We define an ABI for what a material is.
  - Some textures.
  - Some bytes.
- This translates directly to a shader interface.

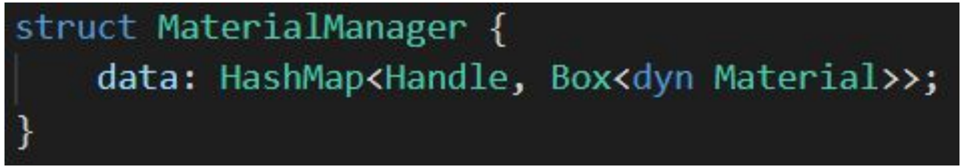


```rust
pub trait Material: Send + Sync + 'static {
    const TEXTURE_COUNT: u32;
    const DATA_SIZE: u32;

    fn to_textures(
        &self,
        slice: &mut [Option<NonZeroU32>],
        translation_fn: &mut (dyn FnMut(&TextureHandle) -> NonZeroU32 + '_),
    );
    fn to_data(&self, slice: &mut [u8]);
}
```

How Do? Structure Iteration 1

Simple solution:



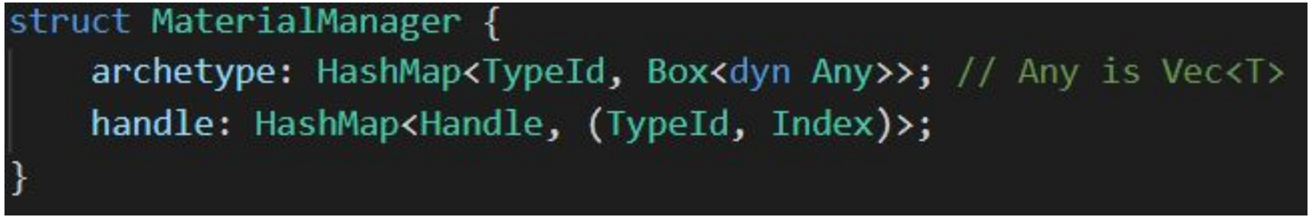
```
struct MaterialManager {
    data: HashMap<Handle, Box<dyn Material>>;
}
```

While this will work, we need to iterate through every single material.

Eat a cache miss *every* time we access a material.

How Do? Structure Iteration 2: Archetypes

The revelation is storing the same type together.



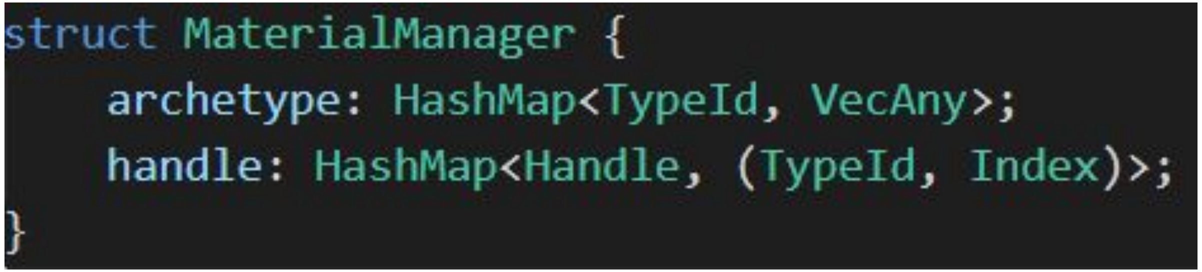
```
struct MaterialManager {
    archetype: HashMap<TypeId, Box<dyn Any>>; // Any is Vec<T>
    handle: HashMap<Handle, (TypeId, Index)>;
}
```

We can iterate through every material an archetype easily. Cache efficient.

Slightly slower on individual lookup due to downcast, but downcast is a comparison.

How Do? Structure Iteration 3: Assisted Archetypes

In addition, we can remove the double indirect of Box<Vec<_>>



```
struct MaterialManager {
    archetype: HashMap<TypeId, VecAny>;
    handle: HashMap<Handle, (TypeId, Index)>;
}
```

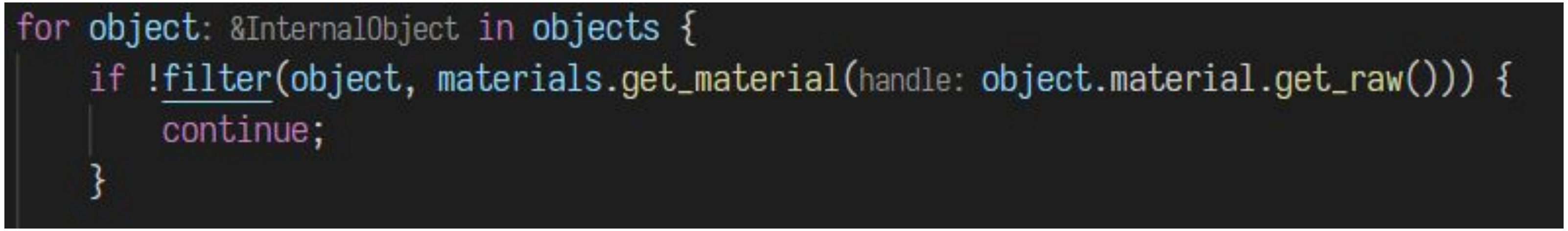
Shout out to @Violet for her work on the list-any crate, providing VecAny, which removes this indirection.

Object Queries

- Render Routines needs to query for all objects they care about.
- Objects are relevant if:
  - They have a material that the render routine cares about.
  - Certain properties of the material are true.
    - Ex. transparency mode
    - Only need this for certain properties of the material.

Object Queries: The Old Way

- We can iterate over all objects, checking if the material type is correct, and if so, reference that check the properties of the material.
    - Requires iterating over every object for every object.
    - Requires checking various conditions for every object.

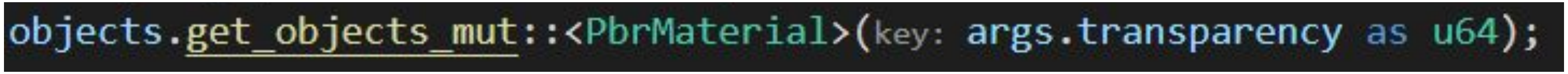


```
for object: &InternalObject in objects {
    if !filter(object, materials.get_material(handle: object.material.get_raw())) {
        continue;
    }
```

- Surprisingly slow, we can do better.

Object Queries

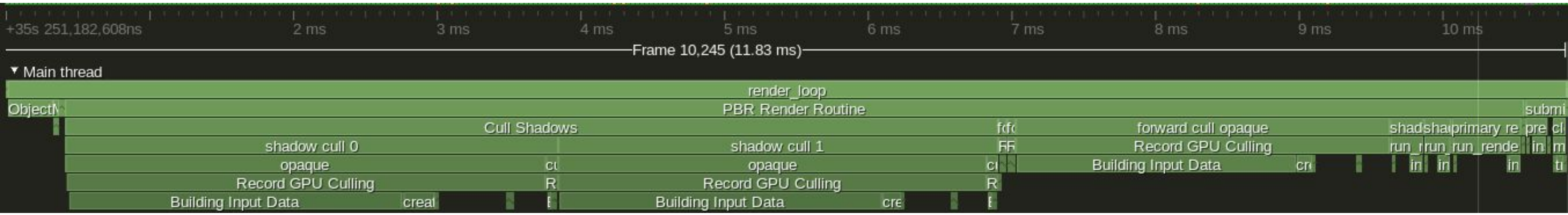
- We can store objects archetypically, with the MaterialType and a u64 material-data as a key.
  - May require executing multiple queries but does not do duplicate work.
  - Allows custom splitting of the object list.



```
objects.get_objects_mut::<PbrMaterial>(key: args.transparency as u64);
```

Archetype Results

50,000 objects, 10.5ms. 1.5x original.



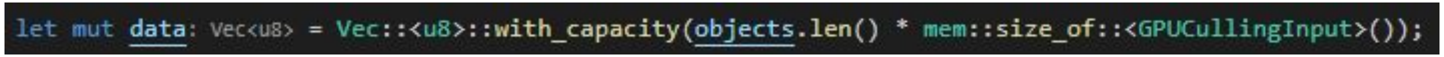
We're doing good, but there's more on the table.

"Building Input Data" still very slow.

Silly Reallocation

There's a heavy reallocation in profile.

This realloc is *very* suspicious. I should be pre-allocating....

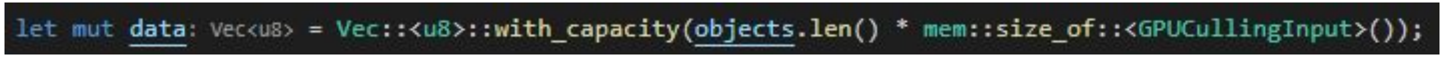


```rust
let mut data: Vec<u8> = Vec::<u8>::with_capacity(objects.len() * mem::size_of::<GPUCullingInput>());
```

Silly Reallocation

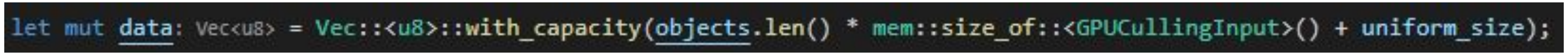
There's a heavy reallocation in profile.

This realloc is *very* suspicious. I should be pre-allocating....



```rust
let mut data: Vec<u8> = Vec::<u8>::with_capacity(objects.len() * mem::size_of::<GPUCullingInput>());
```

I am missing the buffer's header....

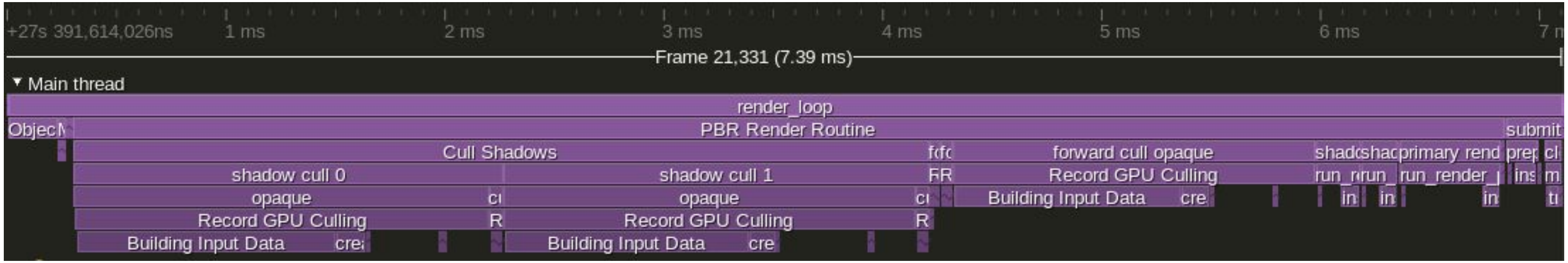


```rust
let mut data: Vec<u8> = Vec::<u8>::with_capacity(objects.len() * mem::size_of::<GPUCullingInput>() + uniform_size);
```

Reallocation gone.

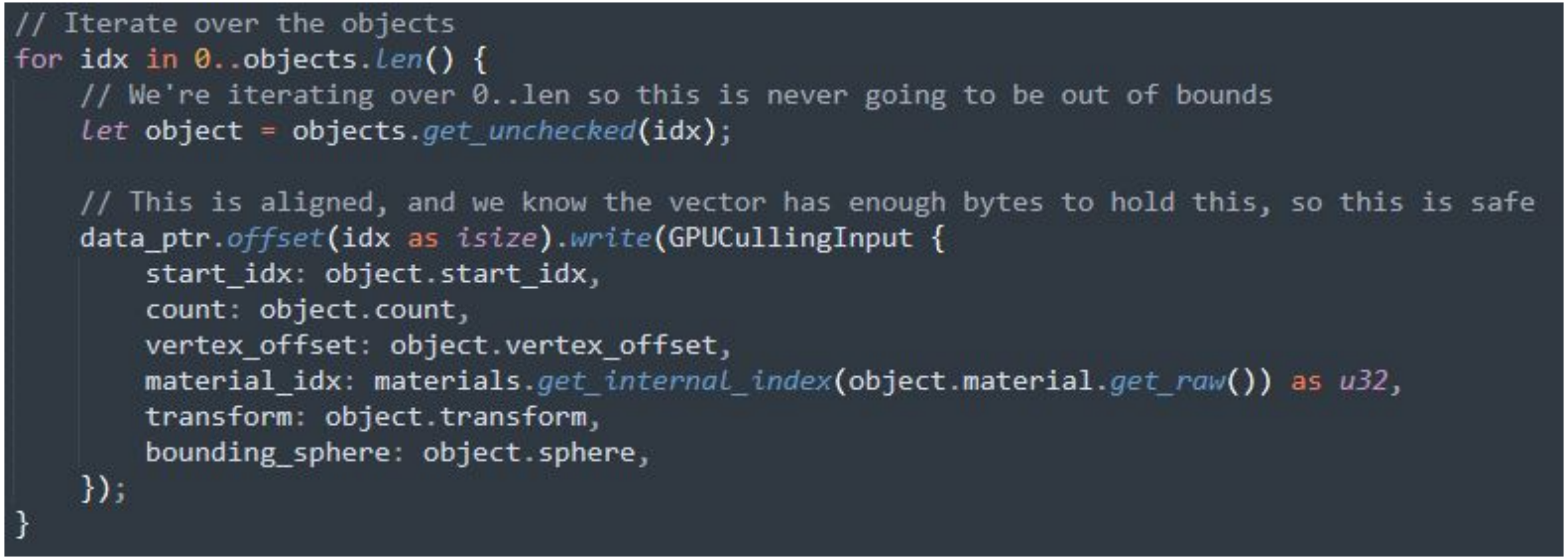
Silly Allocation Results

50,000 objects, 7.1ms. 2x original.



We’re now in the realm of optimizing our memcpys. This memcpy is sitting at ~6GB/s. We can do better.

Improving Our Memcpy



```
// Iterate over the objects
for idx in 0..objects.len() {
    // We're iterating over 0..len so this is never going to be out of bounds
    let object = objects.get_unchecked(idx);

    // This is aligned, and we know the vector has enough bytes to hold this, so this is safe
    data_ptr.offset(idx as isize).write(GPUCullingInput {
        start_idx: object.start_idx,
        count: object.count,
        vertex_offset: object.vertex_offset,
        material_idx: materials.get_internal_index(object.material.get_raw()) as u32,
        transform: object.transform,
        bounding_sphere: object.sphere,
    });
}
```

We have a hashmap lookup in there. Can we eliminate it?

HashMap BadMap?

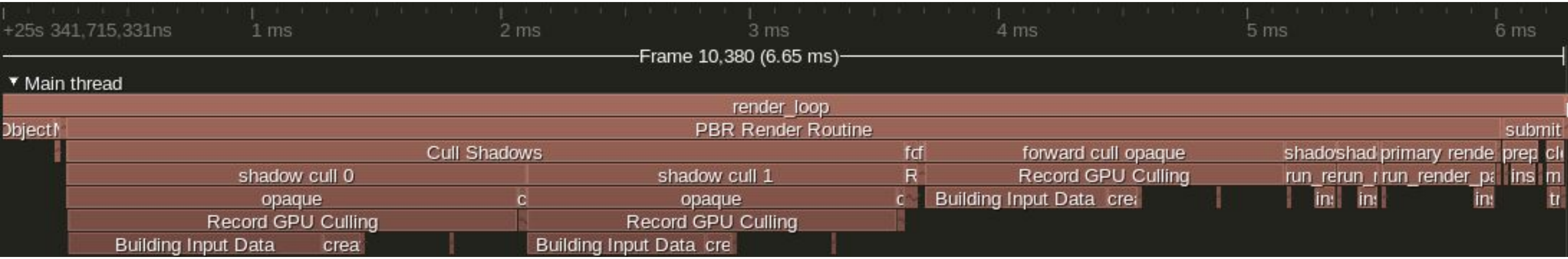
- We are looking up in this hashmap because each object has a RawMaterialHandle. We need a raw index. Handles are stable, but indices aren’t.
- When we have such a tight loop, this single lookup can hurt a lot, even with very fast hashes.
- What happens if we store an index instead of the handle.

Data Denormalization

- We can store the index directly in the object.
- Needs to be updated if the index changes (such as material archetype change).
- Requires storing a Vec<RawObjectHandle> and a for loop updating all objects on material change.
  - This reads as slow, but it's still a contiguous array.
  - Material changes happen 1-10 times/frame. This loop fires 200k times.
- Slightly pecimizes material operations, which are rare, but highly optimizes this memcpy.

Data Denormalization

50,000 objects, 6.3ms. 2.4x original.



Let's look at the assembly.

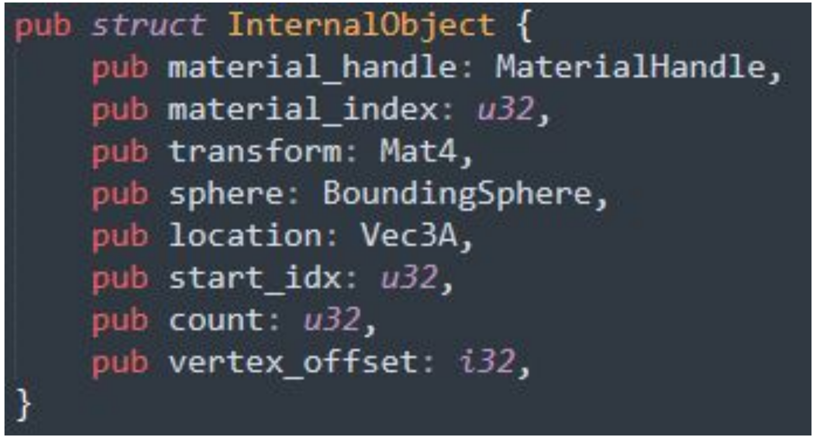
Assembly

❌

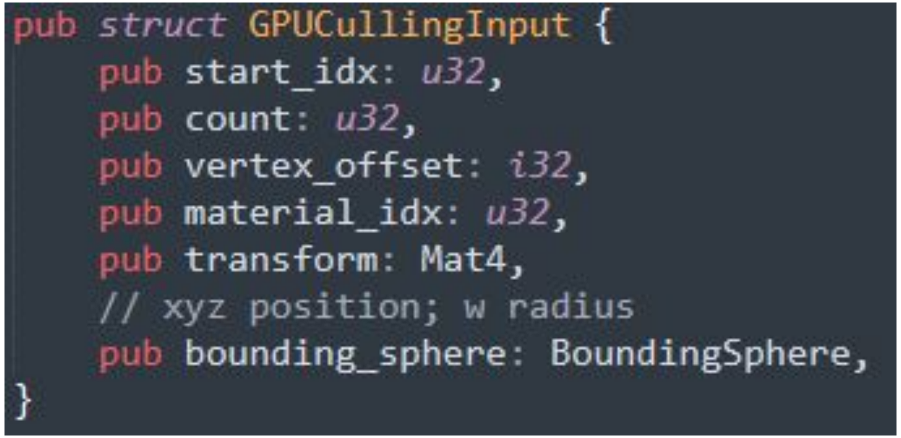
This memcpy loop is doing a lot of shuffles....

Memcpy Dumber

The members need to be shuffled within the memcpy. (Both repr(C))



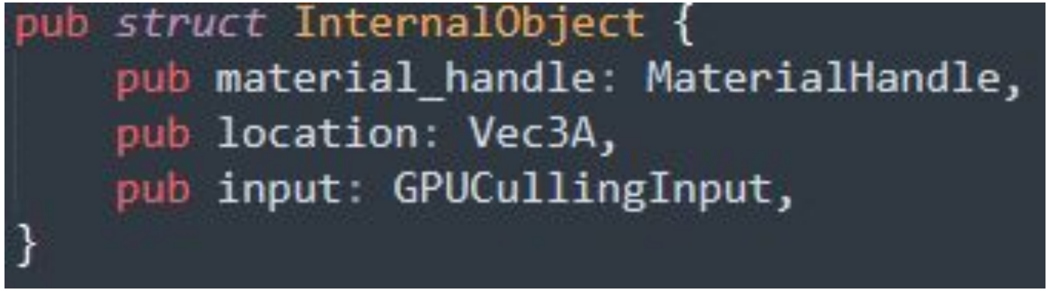
```
pub struct InternalObject {
    pub material_handle: MaterialHandle,
    pub material_index: u32,
    pub transform: Mat4,
    pub sphere: BoundingSphere,
    pub location: Vec3A,
    pub start_idx: u32,
    pub count: u32,
    pub vertex_offset: i32,
}
```



```
pub struct GPUCullingInput {
    pub start_idx: u32,
    pub count: u32,
    pub vertex_offset: i32,
    pub material_idx: u32,
    pub transform: Mat4,
    // xyz position; w radius
    pub bounding_sphere: BoundingSphere,
}
```

Memcpy Dumber

Simple solution: nest the structs.



```
pub struct InternalObject {
    pub material_handle: MaterialHandle,
    pub location: Vec3A,
    pub input: GPUCullingInput,
}
```

Beautiful Memcpy Loop

❌

Just raw read and write of 16 bytes each.

Extra Memcpy

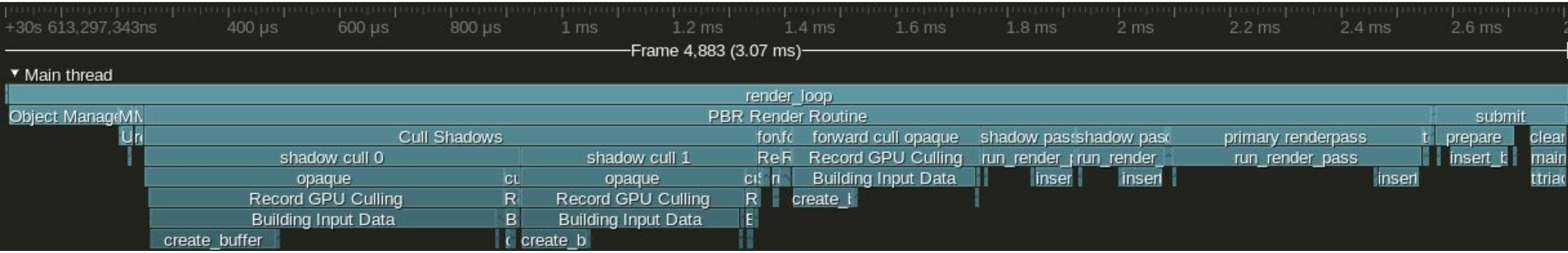
This solution is using wgpu::Device::create_buffer_init which takes a reference to the data and copies it into the internal buffer.

Copy doesn't matter for small or infrequent data, but hurts for large arrays.

wgpu buffers offer a `mapped_at_creation` flag that lets you memory map the staging array immediately. Let's build into this instead of building up a vector to copy.

Memcpy Optimization

50,000 objects, 2.8ms. 5.5x original.



Those seemingly small changes helped *a lot*.

We're now running at ~20GB/s. This is much much better, and very reasonable for a memcpy loop.

Don’t Forget The Dumb Stuff

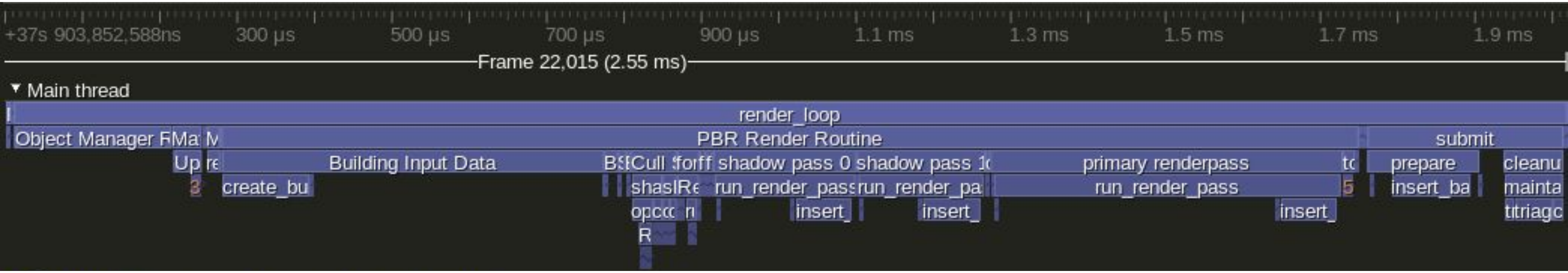
We still have some redundant calculations. We are building and uploading the culling input for multiple times per frame.

We need to cull separately per camera (shadows are cameras), but culling input is the same.

We can do 3x less input building.

Dumb Fixes

50,000 objects, 2ms. 7.8x original.



I am extremely happy with these results. There are ways forward to speed things up, but would require significant overhead to remove buffer allocation and bind group allocation.

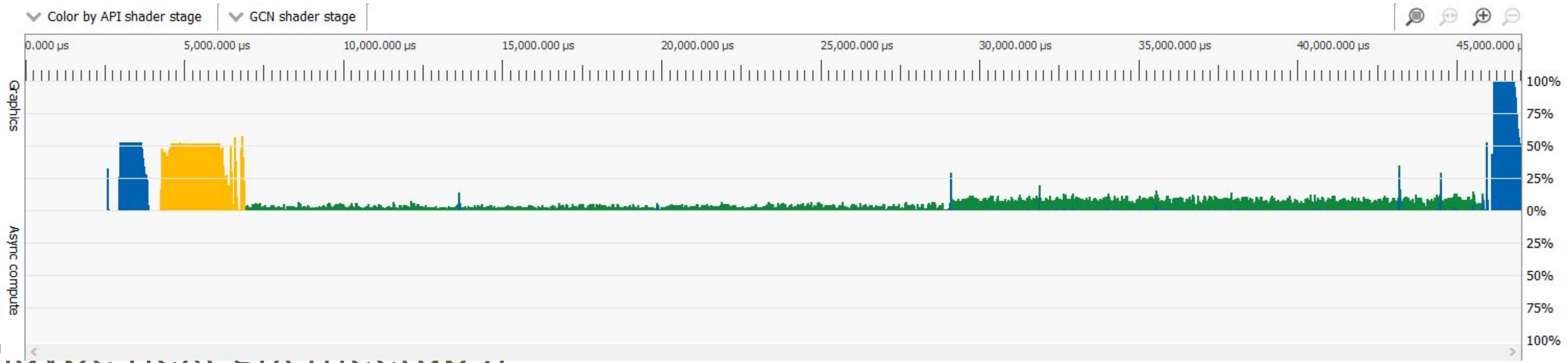
Lessons Learned

- Profile, profile, profile.
  - Profile!
  - Profile!
- Work that I initially thought would be slow, was completely insignificant.
- I could have solved this problem with threading.
  - Heavily consider avoiding threading until you've exhausted most other optimizations.
  - It can easily hide massive slowdowns.
- I could have solved this problem with sparse updates.
  - When full updates are needed, still have to endure the cost.
  - Lots of infrastructure needed.

Part 0b100: Conclusion and the Future

CPU Tackled… GPU Next

We’ve tackled the low hanging CPU fruit. There still GPU optimization needed…



Maybe next gfx meetup :)

Upcoming in rend3 v0.2.0

- New Material/Object System.
- All the optimizations.
- Multi-threadable texture uploads.

Plans for v0.3.0

- Composable Render Routines.
- Egui and Imgui Integrations
- Gpu-side Optimizations

v0.2.0 Release Coming Soon

1 Month Release Cadence

Correlates with Meetup & Newsletter

Acknowledgements

Thank you to my contributors:

- @scoopr
- @noxim
- @MindSwipe

@kvark for ideas about adopting ECS patterns in a renderer.

@John-Nagle for testing, continued support, and spearheading ideas.

Call To Action

There are tons of things to do in rend3. Need all the help we can get.

Feedback, contribution, ideas are all welcome.

If something doesn't work for your use case, come find me and let's have a chat.

docs: https://docs.rs/rend3/
repo: https://github.com/BVE-Reborn/rend3
examples: https://github.com/BVE-Reborn/rend3/tree/trunk/examples

@cwfitzgerald on discord, #wgpu-random matrix