# Monday

## 7:00-7:20 Fill out higher-level goals and work log

## 7:25-8:25, 9:05-9:20,10:00-13:30 Implement OBB

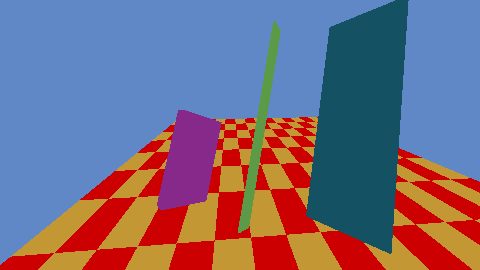
I implemented a YAML parser yesterday but forgot to submit the changes to perforce.

To implement this, I will simply pass a vector containing the rotation around each respective axis. In world space, the object appears rotated. In object space, the object appears axis aligned. When we want to check for intersection, do I transform the object to world space or do I transform the ray to object space?

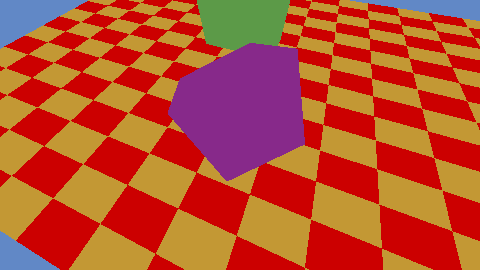
I think that I should transform the object to world space because that is the order in which the book does viewing transformations. This how the book does it but that is for rasterization. In ray tracing, it feels like it would make more sense to do it the opposite way. I also think that it is the only way to do it right now because my code only works with axis-aligned rectangles. This means that I have to transform the ray to object space.

To transform the ray to object space, I have to rotate it. I think that in object space the origin is located at the object’s center. This is only when you use translation in your transformation matrix. I may do that, but I haven’t done that as of now. Changing that, however, should not matter because both variables are multiplied by the transformation so the result won’t change.

**8:05 Rotated AABBs**

[****](https://drive.google.com/open?id=1Q4POImUIlp8bUCYPuIDnkFRZA3JiUn5m)

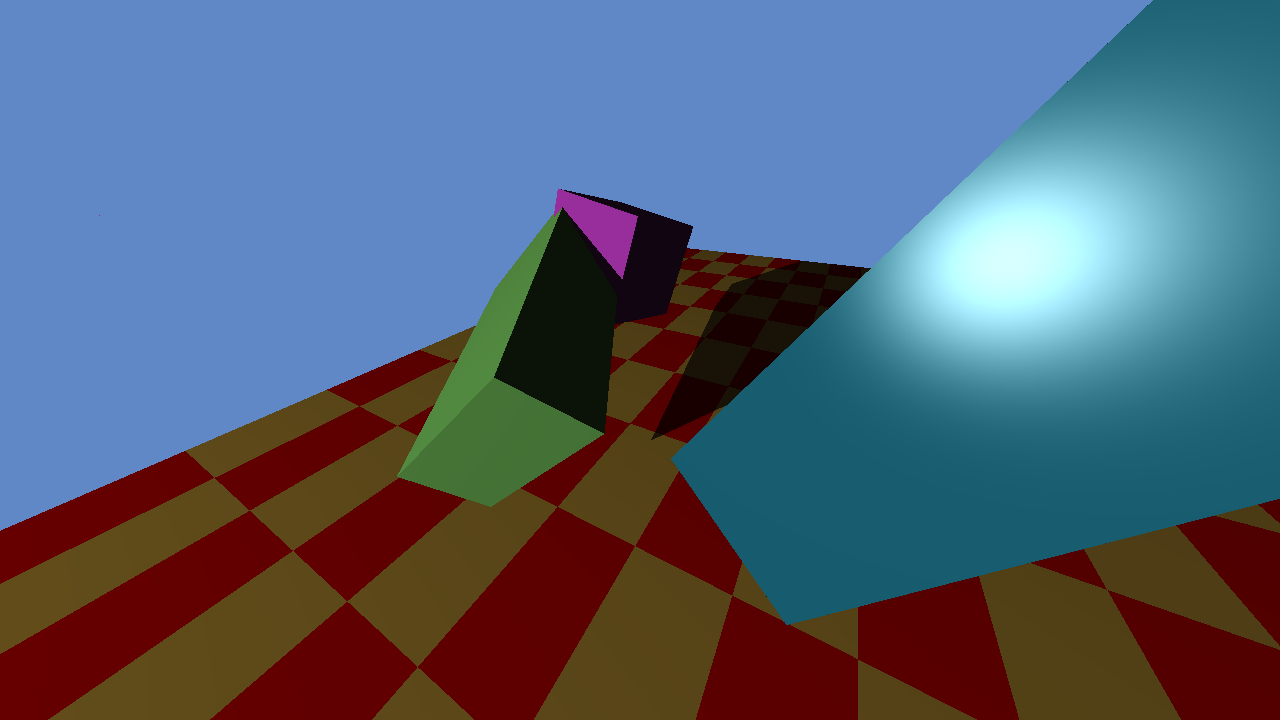
The rectangles lose a few dimensions and the normal calculation doesn’t work, but the result is not too bad. The rotation seems to work partially. I’m not sure at this moment why it doesn’t retain its dimensions and will research into it.



I should probably make a class called OBB for this. The problem was that I was transforming the ray and the object position, but the object position is already axis aligned. If I transform the object coordinates, they are no longer axis-aligned.

This is a bit odd though because in object space the object is supposed to be at the center point or at least a point on the object (not sure about that). Right now, we transform the ray to a basis that is the same as the object’s basis. It’s quite a neat solution, now I just have to fix the normal calculations.

**9:05 normal calculation**

****

I forgot to convert my degrees to radians. I transform my ray to object space. I still use the world origin though so I assume that my rectangles are rotated around the world origin. I will research that thought.

**10:00**

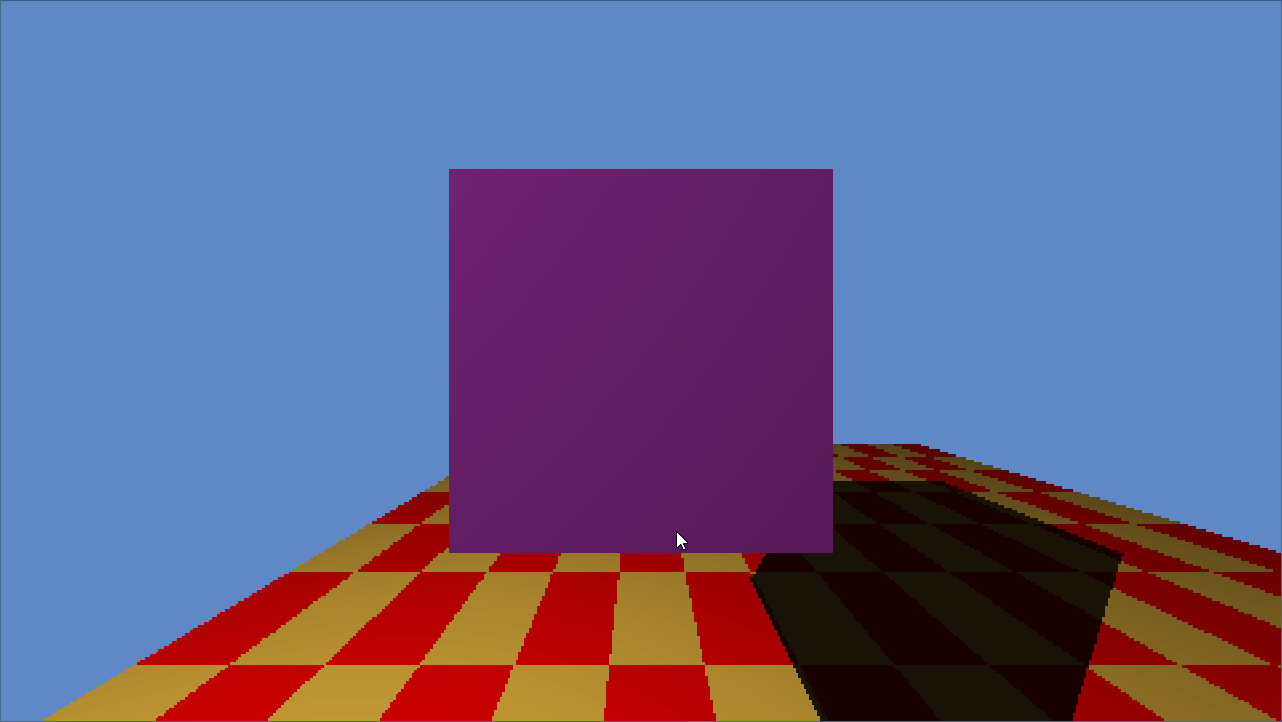
Apparently, my rotations around the x-axis and y-axis are clockwise. This is incorrect and I’m not too sure why. The unit test is also correct because I checked it with my own drawings.

**12:10**

I’m spending way to much time on this.

**14:05**

Here is the problem. I translate my direction vector but I shouldn’t. To fix this, I can either do two things. I can add a vector 4 class so that I can specify the 4th component. Or I can add a function that returns a matrix without the transformation data. I’ll probably add a cheap vector 4 with one function.



AABBs now rotate around their own origin. No longer around the world origin. This indeed took too long. The problem was that my direction also got translated. I thought that there was something wrong with the matrices and kept debugging them. At first, it looked like a problem with the inverse calculation.

## 9:20-9:45 Meeting with Phill

## 13:30-14:15 Help Bryan and David

## 14:15-14:30 Noticed error with AABBs

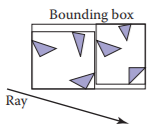
This was such a stupid oversight. The AABB checks if rotation object in JSON exists, if it doesn’t. Is doesn’t change the directory. If it does exist, it goes into the JSON.

## 15:55-17:00,17:50-18:30 Research what spatial data structures are

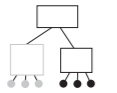
Spatial data structures organize objects in space so they can be lookup up efficiently. The book talks about 3 general classes of spatial data structures:

* Structures that group objects into a hierarchy are **object partitioning schemes**. The groups may overlap
* Structures that divide space into disjoint regions are **scene partitioning schemes**. One object may intersect more partitions
  + Space partitioning can be regular, space is divided into uniform shapes
  + Space partitioning can be irregular, in which space is divided into irregular pieces with smaller pieces containing more objects

The three structures that the book describes are:

* Bounding volume hierarchies
* Uniform spatial subdivision
* Binary space partitioning

**16:15 Hierarchical Bounding Boxes**

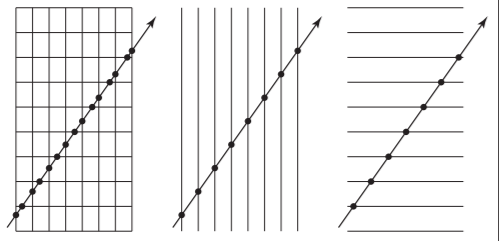
Hierarchical bounding boxes (HBS) are simply boxes that encapsulate objects. You partition these boxes in divide and conquer manner. In memory, these boxes may represent a tree in which a box is with its smaller sub-boxes. The key concept is that we divide objects from each other.

Rays can enter other bounding boxes at all time.

By making an HBS node of surface type, we can recursively traverse through the HBS. The key is that primitives are also a subclass of a surface.

When creating a bounding volume hierarchy, it is convenient to make it a binary tree. Boxes of subtrees should not overlap too much.

You can build a three by partitioning. Just create subtrees until an arbitrary amount. I should research into more ways to create a bounding volume hierarchy.

**16:45 Uniform spatial subdivision**

In Uniform spatial subdivision we have nodes of which objects may be part of. You split spaces into different parts. Objects can belong to many nodes.

We divide the scene into axis-aligned boxes. We check for intersections with the surfaces inside the box or the boxes around it.

Because each box is of the same size, we can easily index a box because of the recurring pattern.

**17:50 Binary space partitioning**

Every node contains two subtrees on either side of the cutting plane. The cutting plane is axis aligned. One side contains all the primitives on that side, and the same for the other side. Primitives that cut a plane are in both subtrees.

There are four cases for a BSP plane. The ray can pass through, testing all its old intersections and after it has passed (if it passed) check all the new intersection or it may go to one direction only and stay in that direction, meaning that only intersection with one subtree has to be done at a time.

**18:25 Comparing different spatial data structured**

I found BVHs and BSPs to be quite similar. They only differ in that one uses volumes and the other uses planes. BVH and USS sound like the most simple ones. I will look into their efficiency, although efficiency is mostly dependent on how efficient the tree has been built.

# Tuesday

## 9:55-10:00 Fill out today’s work log

## 10:00-12:25 Research YAML tutorial

The library uses exceptions for file state. I can use those exceptions in a try-catch block, but I’ve read many articles saying that exception handling is bad because it breaks the logical flow and is almost the same as a goto statement. I will look more into this because I'm not convinced by those reasons.

This [post](https://www.quora.com/Why-do-some-people-recommend-not-using-exception-handling-in-C++-Is-this-just-a-culture-in-C++-community-or-do-some-real-reasons-exist-behind-this) sums up al my thoughts. Exceptions are the most logical way here and I’ll use them as intended.

**11:00**

I decided to use exceptions but I’m having trouble parsing arrays in YAML.

**11:45**

I have cleared up on the terminology used by YAML. Sequences are arrays. Maps are objects.

**12:00**

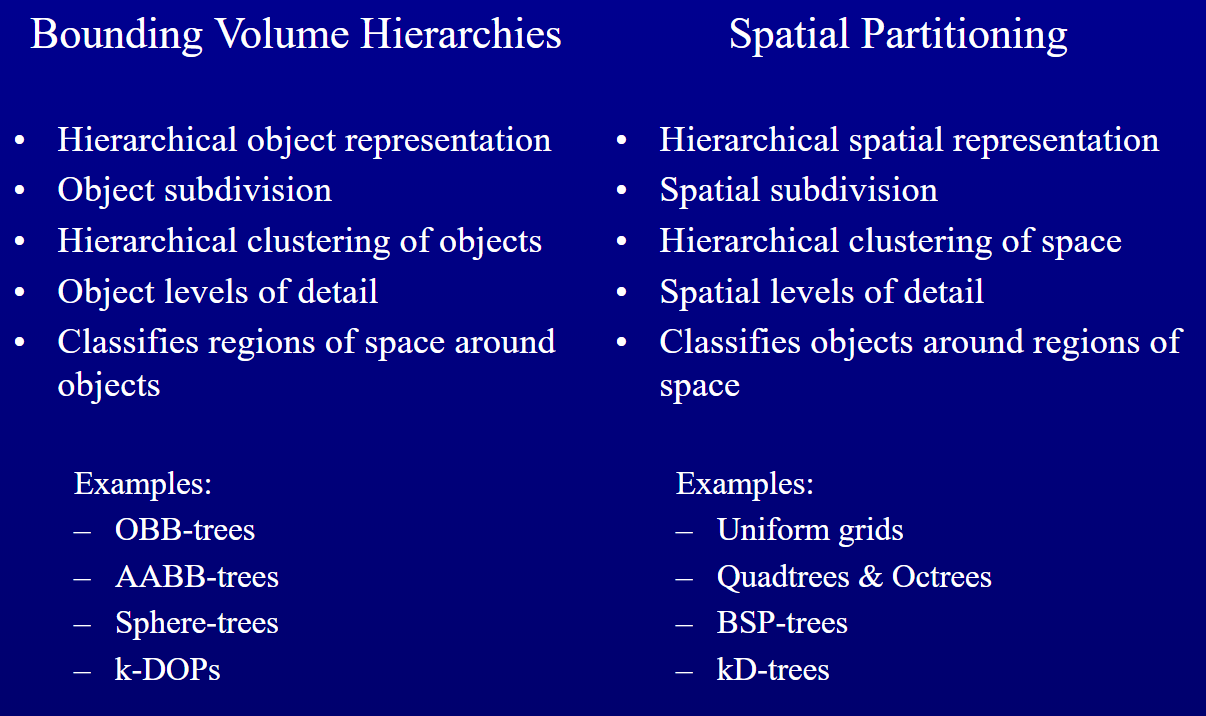
At first, I wanted to implement YAML because it would make it easier for me to edit and prototype files. But right now I’m thinking that a better way to implement reference image 2 is to just make the scene data structure more flexible by writing a function that produces scene 2 instead of making it manually. I enjoyed creating the binaries for the YAML library but I have decided to leave it at JSON.

## 16:30-18:15 Research differences between the spatial data structures

Apparently, the three structures that the book describes for spatial data structures are :

* bounding volume hierarchies
* hierarchical space subdivision (Binary space partitioning, BSP)
* uniform space subdivision

The terms seem a bit loosely. I’m not sure how bounding volume hierarchies and hierarchical bounding boxes differ from each other. Is one the method and the other the implementation? I will refer to their acronyms as methods (BVH).

I think that there are two major differences between the three. 2 dedicated groups. The bounding volume hierarchies and the spatial partitioning.

This powerpoint is a great comparison between the two methods. BVH puts spaces around objects and SP puts objects in regions of spaces. Knowing this, I can make some deductions. Obvious is that the more spread your objects are, the longer spatial partitioning should take and with BHVs, the more objects there are, the more boxes there are.

There is no definite answer to which spatial data structure is the best. You either go with your gut or do a careful analysis. There are a few factors that I do think of that are important for this decision. How complex one is to implement. These spatial data structures are seen everywhere so I should try to learn them .

I’ll look into spatial data structures that are simple to implement (because of the time that I have left) and are efficient for static scenes (fast efficient traversal is most important because my scene is static).

Here are some draw backs for uniform spatial subdivision:

* Voxels (the subdivided spaces) can take up much memory and some voxels can be empty.
  + Make up an algorithm that mixes empty voxels
* Hard to find the ideal size of a voxel
* The same primitive can be tested because primitives can be part of multiple voxels
  + Use some unique ID (extra check, extra memory)

Many of the rectangles in the reference image are touching the same voxels (they are oriented). There is a lot of overlap

Here are some disadvantages for Bounding Volumes:

* The bounding volumes should fit the shapes thight, space should not be wasted. The smaller, the better.

In my opinion bounding volumes seem like the better choice. I think that uniform spatial partitioning/grids are good for scenes where primitives are split up a bit into their own spaces. That is not the case with reference image 2. Everything overlaps. I also can’t really find any benchmarks about the two spatial data structures. I will probably have to test both out and benchmark it myself. The reason why I went with BVHs is because I could find more guides and information about it. For so much research, I’m slightly dissapointed that I can’t find a good benchmark or guideline to follow when making this decision apart from it depends on the project.

## 18:15-19:00 Research how to use Big O notation

It’s a method used to benchmark how long a certain algorithm is working on a set of data. Where O represents the function and n the operations. A simple search would be O(n). That’s all there is to it. I’ll try to

## 20:00-21:45 Benchmark a few algorithms

I’ll benchmark a few algorithms that I’ve written. I’ll benchmark the determinant and inverse of matrices and I’ll calculate the Big O notation of the pixel color calculation.

**20:10 Big O of pixel color calculation**

I will first make up an algorithm for only diffuse colors.

// Diffuse color algorithm

Check intersection with all surfaces

For all light sources

Check intersection with all surfaces

Say that surfaces has size n and light sources has size m. Because we can’t know the size of these variables as they are not constant, We’ll have to use two variables in our notation.

For every n operation do m \* n.

Which meant that it’s O(n\*m\*n) which is **O(n2\*m)**.

If below maxRecursionDepth

Check intersection with all surfaces

If diffuse

For all light sources

Check intersection with all surfaces

Else if reflective

Recursive with refraction ray and +1 recursionDepth

Else if refraction

Recursive with refractive ray and +1 recursionDepth

Recrusive with reflection ray and +1 recursionDepth

This function also has an unkown variable. MaxRecursionDepth as **d**. Because this value is not constant, I’ll also have to add it. Simply said, the function has a chance to call itself 2 times. Because I’m gonna take worst case scenario the notation should look like this:

**O(n\*2(n\*d)\*m\*n)** which is **O(n\*n\*d\*m\*n)** which is **O(n3\*d\*m)**

**20:50**

I don’t think that I’m doing this right. Reading [this](http://www.cforcoding.com/2009/07/plain-english-explanation-of-big-o.html) article, Big O notation does not keep track of time, it represents the relative complexity. Whatever that may be. Big O shows how fast the [runtime grows](https://www.interviewcake.com/article/java/big-o-notation-time-and-space-complexity). Big O also is only concerned with big values, relative complexity meaning when n becomes very large, how does the function change, how the algorithm scales with the variables. Either way, I think that the Big O notiation is not necessary wrong, but it seems to complicated because I’m not too sure what it is supposed to represent. I’ll ask the teachers to check if my notations are correct.

**21:20 Benchmark determinants**

I basically have two function that I first need to know the upper bound of. calcDet and createCofactor. The determinant function is basically a call to calcDet with the correct dimensions, but because the dimensions don’t matter. I can just use a matrix of any size.

// In calcDet

If Dimension is 2

Return value

For every dimension

Mutiply cofactor

// In cofactor

If Dimentions is 2

Return calcDet

For every dimensions is row

If row is i continue // We can skip these but I leave them for clarity

For every dimension is col

If col is j continue

Submat = mat[col + row \* dimensions]

Det = calcDat(submat)

As we can see, we first call calcDet. calcDet calls cofactor for every dimension. Cofactor contains 2 for loops that iter over every dimension.

Say dimension is 3. calDet will call createCoFactor 3 times. createCoFactor loops over 3 2 times. This means **O(n(n2))**. Of course, this simplifies to **O(n3)**. That is the Big O for the determinant.

For the inverse, we calculate every cofactor for every element in it so **O(n\*n(n\*n))**. Then we calculate the det of the matrix **O(n3)**. Adding the two together becomes **O(n3+n4)**.

## 22:15-23:00 Research BVH implementation

BVHs are quite simple actually. We make a class called RayAccelerator. This will serve as the interface. Every RayAcellerator has a build phase. This allows Liskov substitution so we can easily use a different spatial data structure. We probably won’t have the time to do that but it is a nice thing to have.

# Wednesday

## 13:45-13:55 Fill out today’s work log

## 13:55-18:55 Research traversal of BVH

**13:55 Read PDF**

I’m studying this [pdf](http://fileadmin.cs.lth.se/cs/education/edan30/lectures/s2-bvh.pdf). I’m a bit confused how the traversal of nodes should occur. How is a node supposed to know its children? I can’t see how this implementation does that.

**15:10 Read smart pointers**

I will create a UML and describe the reasoning behind it. I will first read a bit about smart pointers because I will try to use them to make my intent clearer. Using smart pointers is good practice because it tells the programmer who the objects is owned by. It makes clear how its lifetime is managed. It also saves the programmer the hassle to deallocate the memory.

**15:25 Completed smart pointer research**

There are basically 3 categories of smart pointers (auto\_ptr gets deprecated so I won’t look at that). **Unique\_ptr** manages the lifetime of an object and deallocates when it goes out of scope. **Shared\_ptr** share ownership and only gets removed if there are no references left to the object. **Weak\_ptr** I’m not too sure about but it can’t dereference the object that it points to. It can only look and see if that object still exists. It can be casted to a shared pointer I think and then you can access it, but I probably won’t need this pointer.

**15:30 Create the UML for the BVH and construction code**

The basic construction loop goes as follows

***// In build (primitives vector reference)***

Create **world BB** by creating smallest AABB that contains all shapes

Create **root node** with **world BB**

Add world node to nodeList

Make **nodeIndex** and set to 1

**buildRecursive**(0, **primitiveSize**, **root node, nodeIndex**)

***// In buildRecursive (leftIndex, rightIndex, parentNode, nodeIndex)***

If **rightIndex** - **leftIndex** <= **primitiveThreshold**

Make **parent node** leaf

Set its **childIndex** to the **leftIndex**

Set its **primitiveCount** to **rightIndex** - **leftIndex**

Set **isLeaf** to **true**

Else

Calculate **splitIndex**

Check largest side

Calculate **splitPos** by getting the middle of the largest side

Sort primitive selection (**left** till **right**) by the side dimension

For each primitive in selection

Move **splitIndex** till primitive pos is larger than the **splitPos**

If **splitIndex** == **leftIndex** or **rightIndex**

*// This happens when one of the AABBs will contain nothing*

*// You could also try to split with different criteria*

Make **parent node** leaf

Else

Create **two BBs** according to **splitIndex**

Correct dimensions of BBs by calculation the best size

Create **leftNode** and **rightNode** and assign the **two BBs**

**parentNode**.makeNode(**nodeIndex**)

Set its **childIndex** to the **nodeIndex**

***//*** *primitiveCount is not important (unused variable)*

**nodeIndex** +=2

Add **nodes** to the **nodeList**

**buildRecursive**(**leftIndex**, **splitIndex**, **leftNode, nodeIndex**)

**buildRecursive**(**splitIndex**, **rightIndex**, **rightNode, nodeIndex**)

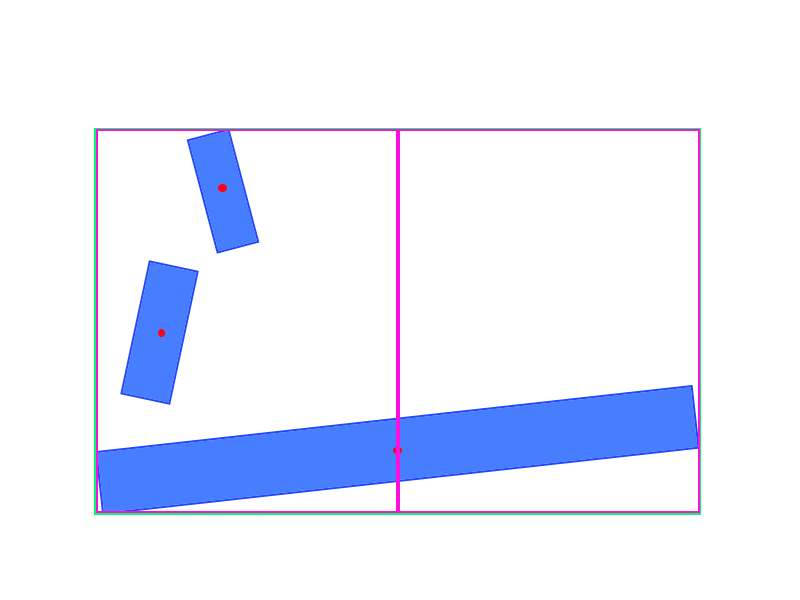
This should do the job. It’s a bit much to explain in words and pseudocode looks a lot more clearly than writing all these sentences. The code first creates the world BB. After that, we recursively create two BBs who create two other BBs.

The new BBs are created with a specific condition. I chose to split through the middle because it seems the easiest to implement. I can’t really say how this is performance wise, I need to test that. But according to the [PDF](http://fileadmin.cs.lth.se/cs/education/edan30/lectures/s2-bvh.pdf) that I was reading splitting through the middle outclassed all other solutions.

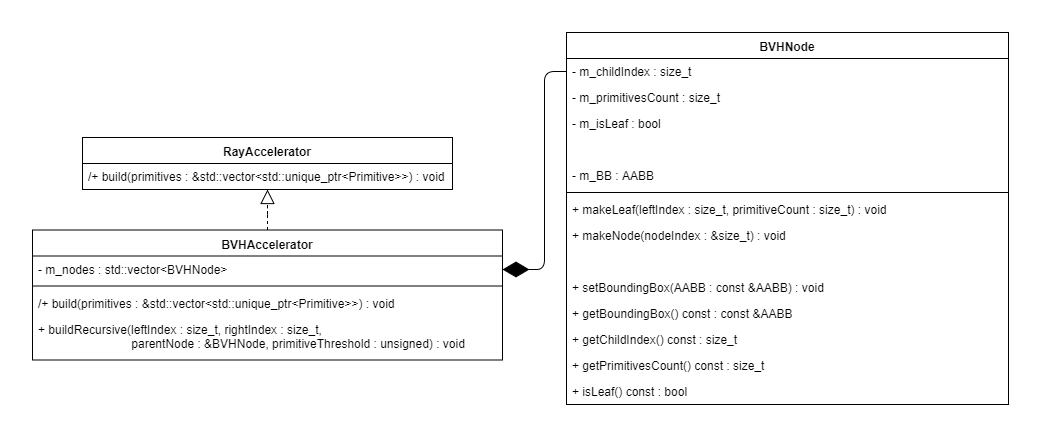
We do this until we reach a certain threshold. The threshold serves as the measurement for how many primitives there at least need to be inside of a BB/node.

If the node is below the threshold, we make it a leaf node, meaning that it points to a section in the primitive list. Else it will become a node, pointing into the node list.

One important thing to note is that we have to check if a new BB even contains primitives. This can happen when a primitive is very long in one dimension and its center lives in the other bounding box.

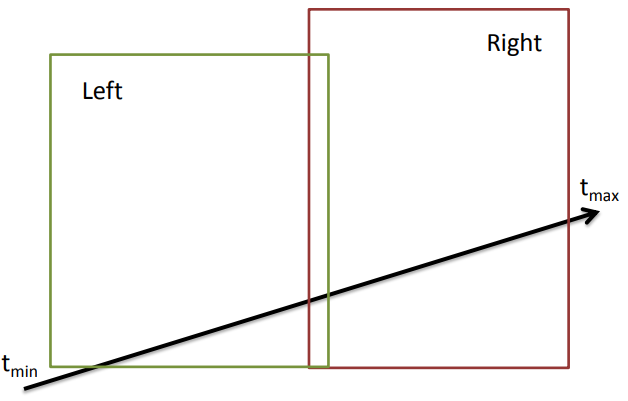
You could say that it is undefined. So just to be sure, the parent node (the green one) will just become a leaf node.

The class diagram looks like this.

[](https://drive.google.com/open?id=15T009VwMURnX8nyUihFGl7Ox__ownuqL)

**17:55 Traverse BVH**

This is all the logic for the node generation. We also need to traverse the node list. To do that, we add an intersection function that takes a ray and an intersection data structure.

We first check if we hit the world bounding box. If we did, check intersection again but with the children nodes

We check both bounding boxes. In this case, both collided. We put the furthest node on a stack and continue checking the children of the closest node.

That’s the pattern. When we hit a leaf node. We check for intersections with the primitive selection it contains (the child index and primitive count). If it contains an intersection, we update the intersection Data with the primitive.

After that, we have to unwind the stack. We have to compare if the unwinded node has a closer hit multiplier than the current intersection data.

That’s all. In pseudocode, it looks like this.

***// In intersection (ray, intersection data, primitives)***

Set **currentNode** to **nodeList[0]**

If **ray** hits doesn’t hit **currentNode**

Return false

Create **nodeStack**

While true

If **currentNode** is not a leaf node

**Intersect** with **both children**

If both intersected

Put **farNode** on **nodeStack**

**currentNode** = **nearNode**

Continue

If one intersected

**currentNode** = **intersectedNode**

Continue

Else

Check intersection with each primitive in **currentNode**

If intersection and **hit** < **intersectionData**.hit

Store **hit** in **intersectionData**

While **nodeStack** not empty

**currentNode** = **nodeStack** unwind

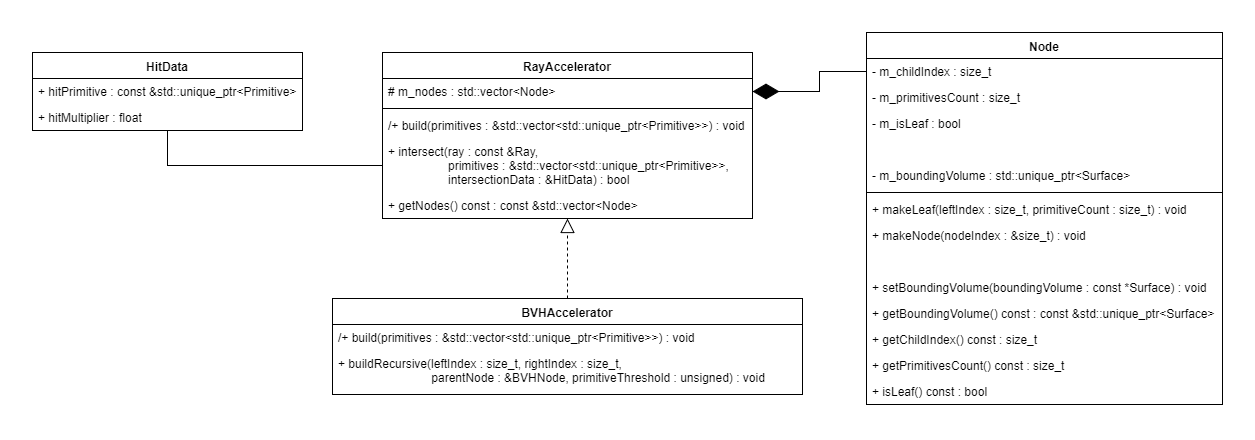
If **currentNode** hit < **intersectionData**.hit

Break

If **nodeStack** is empty

Return **intersectionData**.primitive is not nullptr

That should be all the pseudocode. So now instead of the ray tracer calling hit on every primitive, it will call the intersection of the BVH. It will then return the closest intersected primitive. This makes the final class diagram look like this.

[](https://drive.google.com/open?id=13DZAGMO9TeSvmL0QtobvMO0_8XzFDVOu)

You could actually replace the whole ray accelerator with BVH accelerator. I’m not too sure about spatial partitioning but it probably takes a whole different approach. Notice that I’ve put the m\_nodes inside of the ray accelerator. This is because I want to access the vector when I want to debug my BVH. With this I can iterate over the boxes and draw them if I so desire. This makes it easy for me to see if an the BVH was generated correctly.

That is all for the BVH.

## 19:40-22:35 Create the final design

I’m a bit confused about the difference of association and dependency. From what I read association is when you access a variable and dependency is when the implementation of the class also changes the result of your class. Does this mean that a getter is dependency? Everyone also says that you shouldn’t overuse it, just show the important things. So for my ray class, I’ll use association because it doesn’t change the class functionality.

**21:30 Issue #1 primitive bounding volume creation**

I have a surface class. A surface is an object that a ray can intersect with. This means that all primitives are surfaces. Every primitive needs a BV for the BVH to create its BVs. Should I add a function to the Surface class that gets the surface’s bounding volume or should I make it a different class?

I will probably add a virtual function called create bounding volume. This kinda bloats the interface (because ray tracer doesn’t need to know the bounding volume, only if it intersects) but it doesn’t add any member variables to the class that are only used during construction of the BVH.

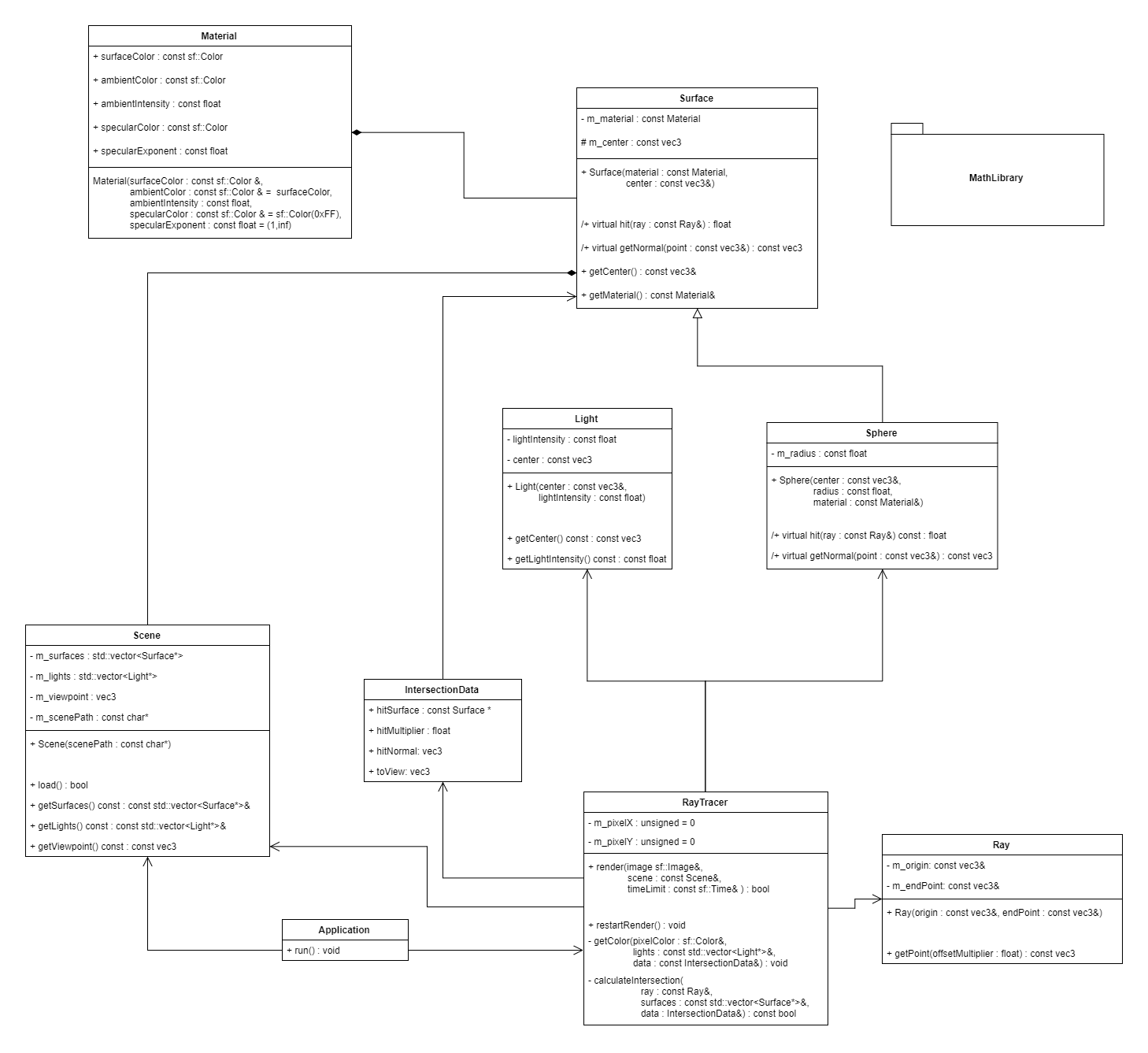
**21:55 Issue #2 how should I split surface and primitive from each other**

The BVH needs AABB to calculate its boxes. Ray tracer needs AABB to calculate the surface color. We have two classes, Surface and Primitive. Primitive is a Surface but contains material data so that we can render it. If the BVH and the Ray tracer both need AABB, should it be a primitive or a surface?

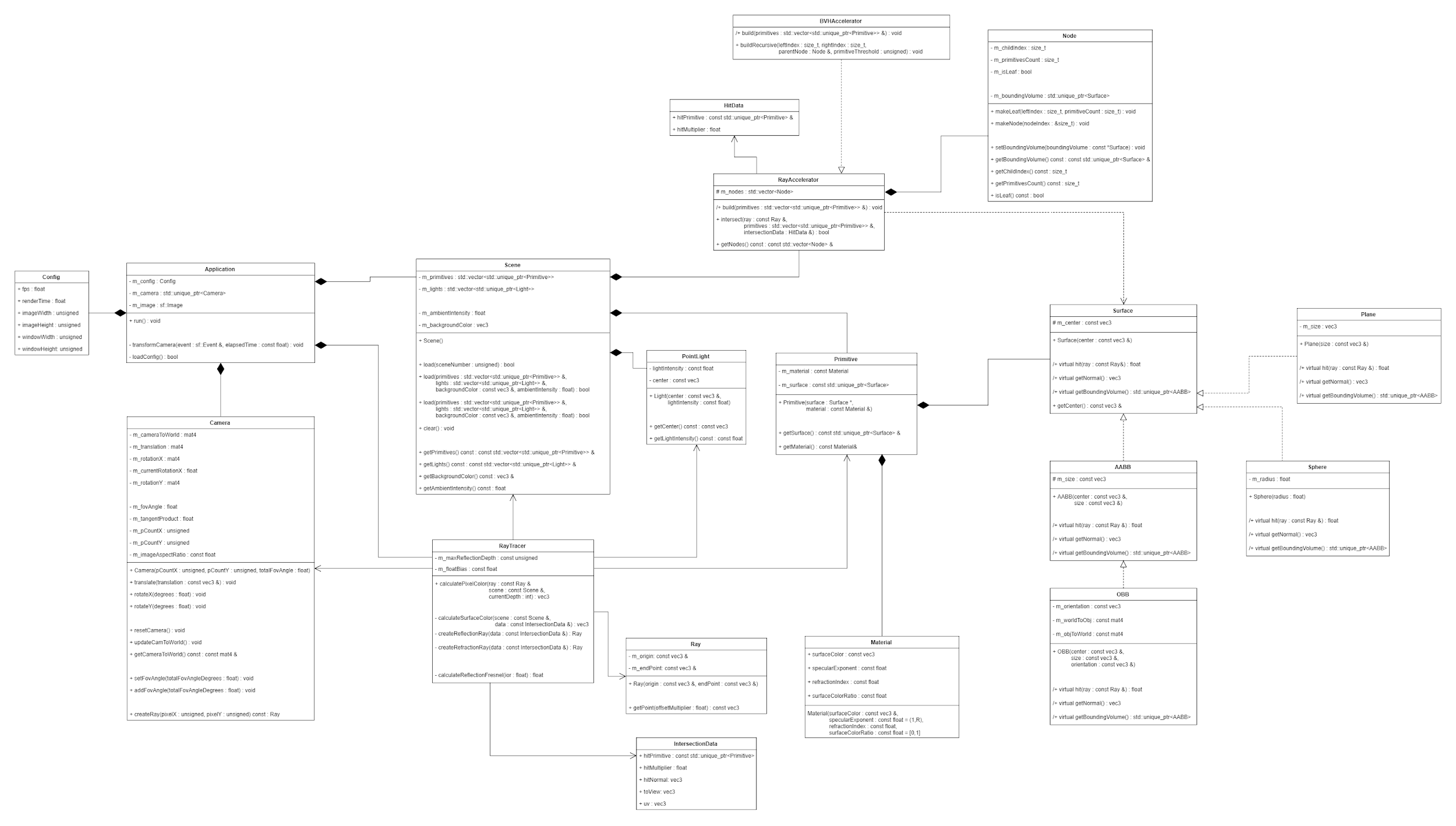
Originally, I wanted primitive to have a bounding volume. It just takes an surface which makes it very flexible. The problem with this is that we can’t calculate the normal. Primitive doesn’t know what kind of surface it holds.

I’ll probably add the calculateNormal function to the surface class. While the BVH doesn’t need it so it may seem like bloating the interface even more. But it actually makes sense. Surfaces are objects that the ray tracer can intersect with. You could actually say that surfaces are objects that the ray tracer can interact with. This means that it should also be able to calculate the normal of any surface.

**22:30 Its done**

That took quite a while, it went from this:[](https://drive.google.com/open?id=1-pwIv7sfrdmrULFkMje11YWM__Om3nbJ)

To this:

[](https://drive.google.com/open?id=1dE20w7baURlwHIXI6tNAL-1nc8rgzAP4)

Wow it become HUGE. I would love to explain it all and I will, but it’s 22:35 right now which means I’m 2 hours over sleep time. I’ll continue this tomorrow.

# Thursday

## 6:45-6:50 Fill out today’s work log

## 6:50-7:10, 9:10-0:00 Justify class diagram

(note: image above goes to original class diagram)

I was working on coloring this image but I actually don’t see a reason to explain it this way. I will explain most of it in comments inside of the image because it makes it more clear. Although I’m not sure if comments are intended for that. I won’t do comments, I’ll make little screenshots of parts of the design and explain the reasoning behind them. First and foremost.

**Surfaces and primitives, how are shapes represented?**

<https://drive.google.com/open?id=1dE20w7baURlwHIXI6tNAL-1nc8rgzAP4>

I’ve decided to write all this documentation in a different file. Also, I can explain all this myself. I rather implement everything before I’m gonna explain it all so that I can at least show a working product.

## 7:30-8:15 ,10:00-13:00 Refactor ray tracer phase 1

**7:30 Add build events**

Right now, the executable uses the project directory as a working director. This is a variable in the project settings. I’ve made the project so that it is easy to distribute the binaries, but when if you run the ray tracer from the binaries, it will of course not find the files. It’s working directory has changed. So the first thing that I will add are build events. I will add a post build event that copies the resources into the binaries folder where the executable is build.

The command [xcopy](https://www.windows-commandline.com/copy-directory-command-line/) allows us to copy one directory to another. I’m not sure if the flags have to be added but I will just in case.

**8:05**

For some reason Visual Studio created iobj and ipdb files in the output directory. I could delete these files with a post build event, but that seems like to much trouble. It is apparently used for [Incremental Link Time Code Generation](http://blogs.msdn.com/b/vcblog/archive/2014/11/12/speeding-up-the-incremental-developer-scenario-with-visual-studio-2015.aspx). It speeds up the whole build process which is nice. I’m not gonna complain about two files if it speeds up build time, but it would’ve been nice if these file went to the intermediate director.

I’ve also added Deploy configuration. This is basically a Release configuration but doesn’t generate a Program Debug Database. I use it when I want to share the binaries with someone (which would’ve been an easy copy paste job but LTCG doesn’t think so).

**10:00 Refactor**

Quite a few problems with getting the template to work.

## 13:00-16:20 Add new method to load scene

Add switch to toggle between bounding boxes.

**14:50**

## 16:20-17:00,18:05-20:00 Implement BVH

I’m having unresolved external error problems. For some reason every function that I call in the class BVHAccelerator gets the linker error

I accidently deleted the BVH folders…

# Friday

## 12:00-15:30 Implement BVH traversal