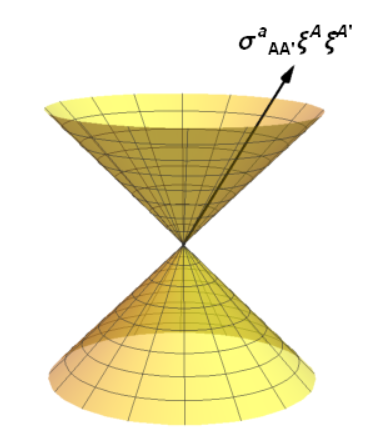
**ICS4U**

**Fall 2021**

**Software Design Document**

**General Relativity**(and other theories of gravitation that is expressible in the framework of Pseudo- Riemannian Geometry) **Visualization Project**

**Zhihao Xu**

**(Code explanation, Source code, Alpha Beta testing results)**

**Jan 30, 2021**

# Table of Contents:

[**Table of Contents:**](#_4l9d752cwto9) **2**

[**Ideas and Program Introduction(Abstract)**](#_x4qxyu5z3yjs) **2**

[**Code Explanation**](#_z87crq6g27ey) **5**

[**Main.java**](#_rzq54xw5dl97) **5**

[**Tensor(package Tensor)**](#_c2utb37knx3o) **5**

[Tensor.java](#_w5trld63ngo9) 5

[Core(package Core)](#_okujt0nqwrjs) 9

[Core.java](#_ejsvq2504vg2) 9

[Math(package)](#_91mc1cuz4xu1) 10

[Nfunction.java](#_cnwe9uu245t) 10

[Ntensor.java](#_ueum9klfnx0v) 12

[Bfunction.java](#_uprzx2jammob) 13

[Numerical.java](#_vduq40gc9cc2) 13

[WColor(package)](#_d5re5v36n5pf) 14

[WFuntcion.java](#_evrxl55m9yj) 14

[WColor.java](#_2ly6l62247r) 14

[WColorFunction.java](#_b25ne9dp7fe0) 16

[Format(package)](#_m5w7d155hd6i) 17

[ObjFolder.java](#_vrdktyhi01m9) 17

[Obj.java](#_e1kqma22vimj) 19

[Region.java](#_u6rjhnk4cls7) 23

[GUI(package)](#_bkoc8hphmvkt) 26

[GUI.java](#_1w9fejvfwj7x) 26

[Engine(package)](#_owwf7jiot7zw) 27

[**Appendix A: Source Code**](#_40c6bwqverog) **27**

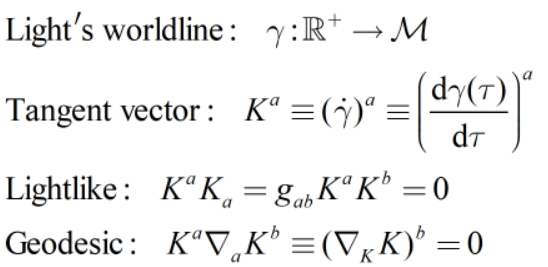
[**Appendix B: Alpha/Beta Testing**](#_uzqwlhm4bgq1) **27**

# Ideas and Program Introduction(Abstract)

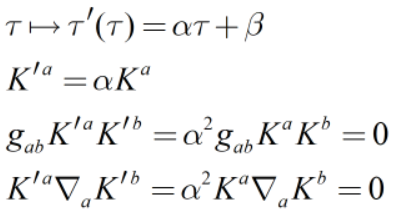
This program is aimed at rendering the actual image of what a camera or observer with mechanics similar to human eyes would see in an arbitrary curved spacetime.

The current realized idea of render is to do all of these by instance ray tracing:

Consider physically: each pixel on the screen is corresponding to a small part of light rays coming from the past objects that travel forward in time and hit the observer’s world line at a certain time in spacetime.

The normal ray tracing (obviously built in newtonian spacetime view) mechanics is to pre-calculate the trajectories of all the light rays emitting from all light sources and change the look of other objects as light hits them. This information will be stored as shader files, which will be used to render users’ screens in real time. The advantage of this method is that users don’t require the same amount of calculating for each frame in game, which gives the possibility of ray tracing in 60 fps with modern computer setup. But for a realistically render in a curved spacetime, the space is no longer static, making the process of pre-calculating and the size of the shader files growing pretty rapidly.

That’s why I chose to do ray tracing for each camera configuration as needed. How exactly will this be realized? Recall that the light in spacetime is traveling with trajectory(worldline) being lightlike geodesic:

We can do arbitrarily bijective affine transformation to the parameter, and all the properties of the light’s trajectory wouldn’t change. If one set α in this instance as -1, it inverted the lightray’s propagation orientation, but at the same time it doesn’t change its trajectory, meaning it coincides with the forward propagation. This is the time inversion symmetry of the light. Tracing forward is the same as tracing backwards. For this reason, we can no longer relying on pre-calculate how each object would illuminate, but instead going backward in time:

For each pixel on the screen, from discussion in the third paragraph, corresponds to a set of lightrays close to each other in direction(by the observer’s celestial sphere induced topology). With inversion invariance, we can replace them with a backward propagating lightray with opposite direction, trace back to each possible object that illuminates to the observer and record the light emitted by them as lightray hits them, eventually displayed on the observer’s screen.

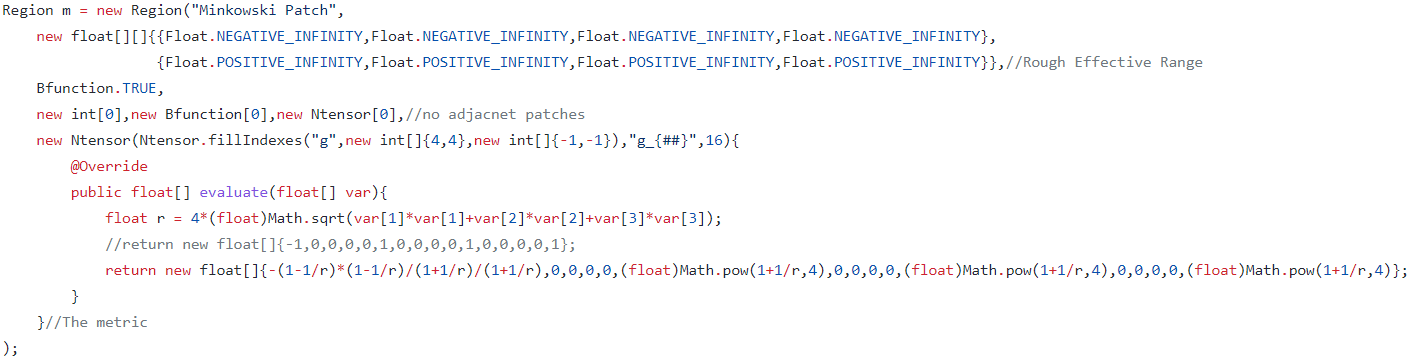
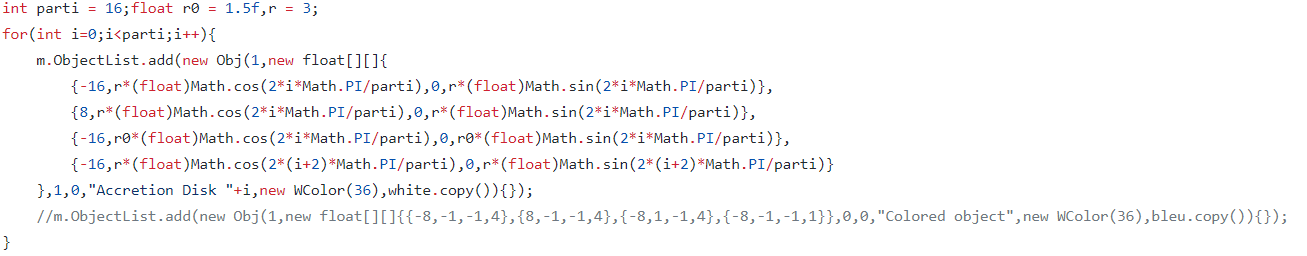
Of course, this approach also has a lot of disadvantages. Calculating geodesics with the same amount of pixels on screen each time is an absolutely hard task for any home-made computer of this age. And for the possibility of even rendering such an image, the accuracy is also redacted a lot. In the current stage of program(Jan 30, 2022), it requires 50 seconds to render a 100\*100 image with max #steps of 100 and order 0 Runge-Kutta method(i.e. Euler’s method) in 5 parallel threads occupying cpu 100% most of time, and 1.9 hours to render a 1000\*1000 image with 2 calculation threads running.

In the future stage of the program, I will possibly further increase the algorithm of the program by the idea I recently thought of: not calculating the lightray for each pixel, but instead using sample lightrays to fill the spacetime and interpolate between them to extends the covering area, thus reduce the amount of calculations, pullback these object to the screen this way.

The initial requirements and design are seen at [GRV - concept/project plan](https://docs.google.com/document/d/1a8Mw3uPUZQ7qacW7xMVBVynJAVC8igiMaP5StQYRPVE)

# Code Explanation

## Main.java

This is the place where things are assembled. Any spacetime configuration:and object configuration:are created here.

And any rendering is also initiated here with observer’s coordinate and frame:

## Tensor(package Tensor)

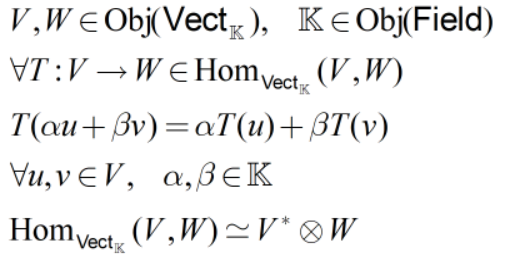
### Tensor.java

**Section 1: preliminaries**

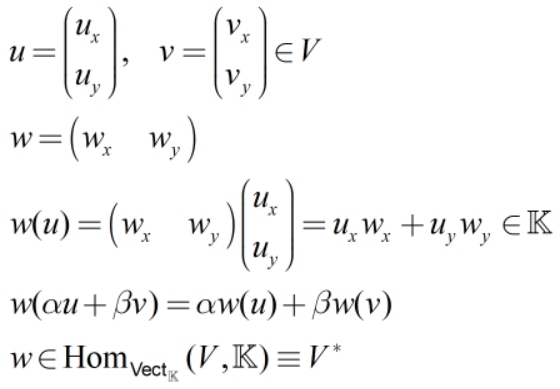
Tensor, mathematically speaking, is a multilinear map(linear to each input slot). Our usual used linear algebra objects can all be unified under this frame.

We can introduce the [Abstract index notation](https://en.wikipedia.org/wiki/Abstract_index_notation) to clarify their difference and make expression simpler. It gives each vector space their abstract index(usually is latin letters, to make difference from component index which is in greek letters.) as their identifier, and their dual space with the same index type but on the opposite side. Contraction (In component form is summing over indexes that are distributed in separately slot pairs, the sum sign is usually omitted with presence of Einstein's summation notation)is represented using 2 abstract index(one is at the top, and the other is at the bottom) that uses the same letter.

I.e. A linear transformation on a vector space is a tensor:



Consider the space of column vectors as the vector space V, the row vectors contract() vectors in V from the left side to give a scalar. It is also a tensor(an element of V’s dual space):

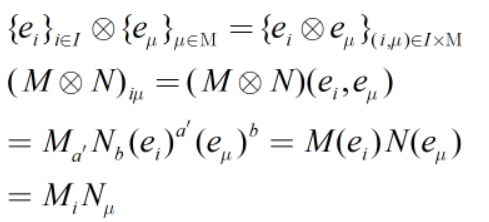
****

The space V itself is also a linear map considering the natural action on dual vectors, thus being a tensor. It can also be viewed as a linear map on the underlying field where it inputs a number in the field, outputs a vector as their scalar product.

A bilinear form is also a tensor as both of the slots are all linear



Inner product of vectors is also a bilinear form thus being a tensor



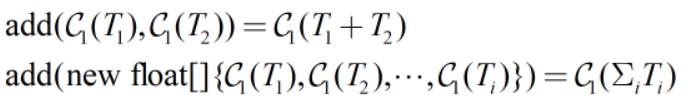
In this program, they are regarded as *float* arrays that are the flattened multidimensional array(tensor product property and basis representation), with an additional *int* array representing the dimensions info of the original tensor. Methods with 4 in the back correspond to the specialized methods for the tensor product of 4-dimensional vector spaces(spacetime tangent vector spaces and their dual). Methods with -i suffix represent the corresponding change in their index arrays.

Let  be the functor mapping tensors to our representation, and let C1 represent the component float array part, C2 represents the index int array part. The equal signs in the following section would be “ideally equal” or equal with float operation approximations

**Section 2: important algorithm explanations**

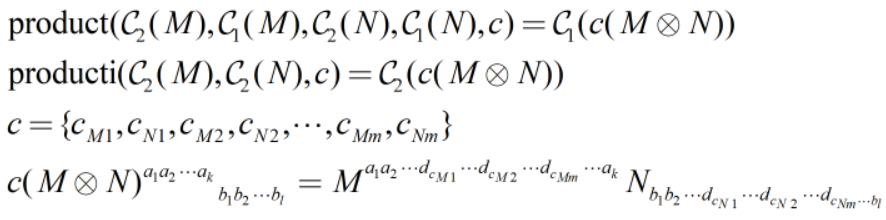
*add(float[] T1,float[] T2):* adds the components of 2 arrays (T1 T2)

*add(float[][]{T1,T2,...,Ti}):* adds the components of i arrays (T1+T2+...+Ti)



*product(float[] M,float[] N):* gives the tensor product of 2 arrays (MN)



*product(float[] Mi,float[] M,float[] Ni,float[] N,int[] c):* gives the tensor product of 2 arrays (c(MN))****

* Creating a int array indicate the current position in the result tensor, and another 2 for position in M and N
* Iterate to add over all contracted indexes
* The pos array also increases 1 at each time, util one reaches the dimension of that index slot, in this case, it clears up the current pos and made increment to the next index
* Loop over all posm posn(thus pos), giving the result

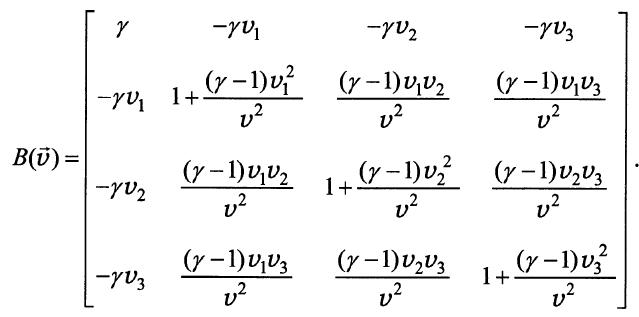
*contract(float[] M,float N)*: gives full contraction of 2 tensors M(N)

*transpose(float[] Mi,float[] M,int[] permutation)*: transpose with indexes doing the permutation action giving(in the form where the kth component in the permutation array will be sends to the component indexed by the value at kth slot)

*scale(float a,float[] T)*: do a scalar multiplication on the given tensor(i.e. Scale each component of the tensor by a)

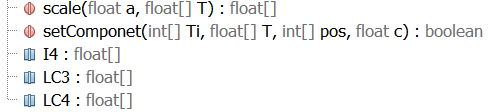
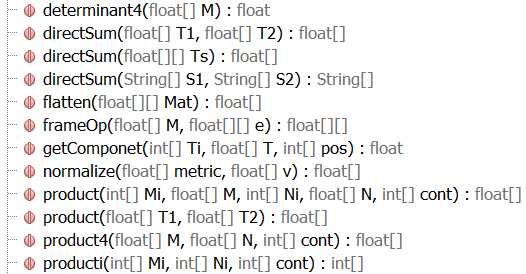
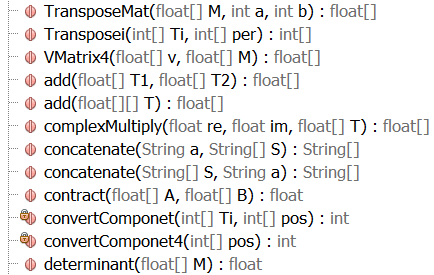
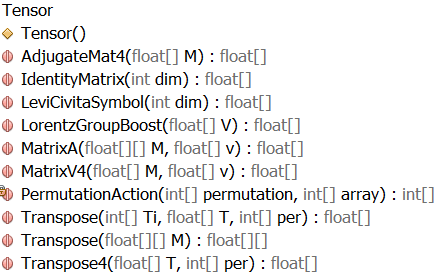


*LorentzGroupBoost(float[] v)*: Do a lorentz transformation to a frame that is moving relative with (space 3-component)velocity v



*LeviCitivaSymbol(int dim)*: gives a rank (0,*dim*) tensor(in fact a rank -1 [density](https://en.wikipedia.org/wiki/Density_on_a_manifold)) in a dimension *dim* vector space, which is fully antisymmetric, and component being 1(when indexes permutation are even), -1(indexes permutation are odd) or 0(indexes not forming a permutation(i.e. Has repeated number)), called [Levi-Civita Symbol](https://en.wikipedia.org/wiki/Levi-Civita_symbol)

*LC3* and *LC4* are static arrays which are the Levi-Civita Symbol in dim 3 or dim 4 for efficiency of the program. In the following text, we use ε with indexes to represent such symbols.



## Core(package Core)

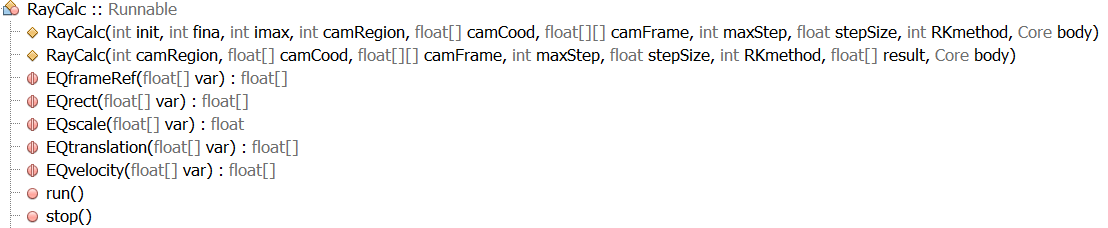
### Core.java

Class Core is responsible for recording the basic information of the observer, configure parameter of the lightray simulation, the screen result data & GUI, and the threads that runs the calculations

Class RayCalc is responsible for a single calculation thread of lightrays. It has 2 separate modes, one is for debugging(showing an individual lightray’s trajectory in spacetime), and the other mode is for normal rendering(the constructor that requires more parameters). It loops through all lightrays that are needed to calculate in the current thread.

The core of the program is in the *run()* method implemented from the class *Runnable*

* At the start of each new lightray, it initializes all the parameters of the lightray(amplitude, velocity, coordinate and manifold patch, etc.)
* Then it uses the Runge-Kutta method given to iterate the configure of the light, evaluating it backwards in time.
* It checks all the objects(object folders) which their envelope cube that covers the current coordinate of the ray, and do a hit test to each object satisfying such requirements
* If it hits one of them, the lightray will add up the amplitude spectrum of light it emitted and the spectrum filter info to itself
* Then the light ray gets randomly get reflected, passed or obstructed with the given change from the configure of each object(*reflection*, *transparency* in class *Object*)
* \*It checks whether this lightray is evolved outside of the current manifold covering patch range, if it does, switch the patch number and do a transformation to the ray parameters
* Repeat the process for *maxStep* times, then switch to next light ray util the work given is complete



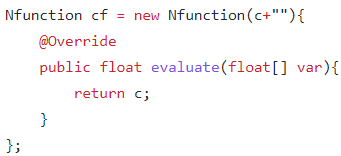
## Math(package)

### Nfunction.java

In our program, we need various kinds of mathematical functions that can be passed to other methods, manipulate(i.e. Having derivative, gradient, add, multiply operations), such as our spacetime structure which expressed in coordinate is usually function of coordinate(transformation function to other coordinate patches is such an example, and the component of the metric, or having the Levi-Civita(metric compatible torsion free) connection(derivative operator)). To satisfy the above requirements with high numerical accuracy at the same time, I made a new *class* called *Nfunction*.

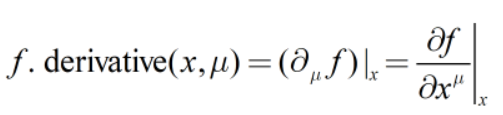
The construction behind Nfunction is simple: it is just a class consisting of 2 things: the latex of the expression, and the evaluate method, which after receiving the parameters *(float[])var* will return a *float* number.

An example of a Nfunction:

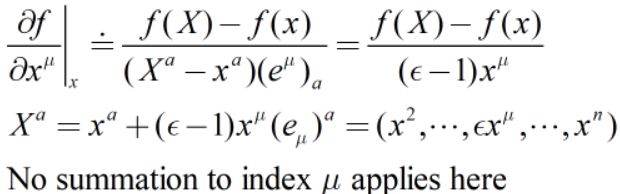


*evaluate(float[] var)*: outputs the value of the function at the point *var*

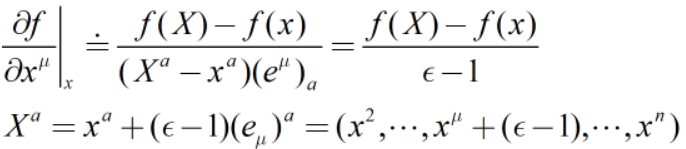


*derivative(float[] var, int idx)*: outputs the derivative respect to the *idx* th variable at point *var*

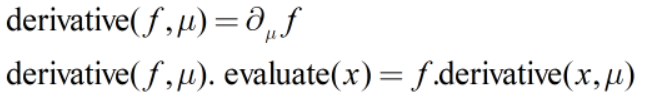
* fix a number that is greater but close to 1 in the float expressible level(*epsilon* ε, *static* value in *class* *Nfunction*)
* If the designated variable (x^idx) is greater than 1, in the extreme case could be 1.15×10^77, when take the usual definition of derivative, the added up “infinitesimal” value wouldn’t even affect the value if it is smaller than 1, making the whole derivative remains undefined(/0 error). For this reason, we take the derivative as the following:



* If the designated variable (x^idx) is smaller than 1, we take our usual definition of derivative to numerically approximate them to avoid singularity of the above algorithm at origin



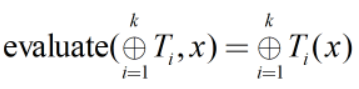
*derivative(Nfunction f,int idx)*: gives the Nfunction form of the derivative of f respect to the *idx* th coordinate

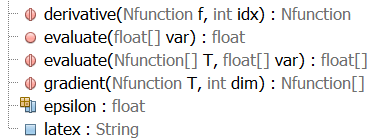
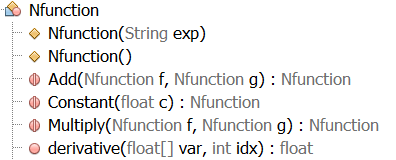


*gradient(Nfunction f,int dim)*: gives the array of Nfunction which includes its derivative respect to the first *dim* variables



*evaluate(Nfunction[] T,float[] var)*: gives the array of Nfunction T’s value at var as float array(embed the direct sum of Nfunction spaces into their fiber product and pick value over a specific fiber)





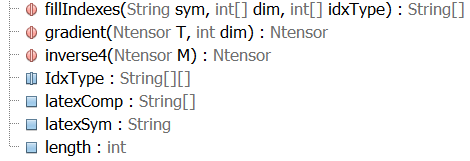
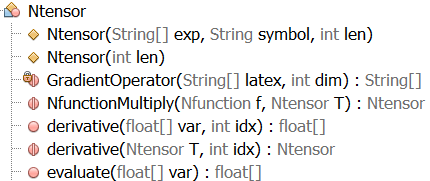
### Ntensor.java

I was once naively thinking: to represent tensors whose value depends on coordinate, using arrays of Nfunction would already be enough. But certainly I was wrong(or I wouldn’t have written them here in the first place). The expression was too long, too lousy, not easily readable and not easy to manipulate. To just represent a 16 components metric with constant value already requires a lot of copy and paste resulting in an extremely long expression.

The methods here are almost all the tautological extensions of operations that are there for a single Nfunction.

*inverse4(Ntensor M)*: gives the inverse matrix of the 4\*4 matrix M pointwise with methods defined in *class Tensor*(*determinant4* and *Adjugate4*)

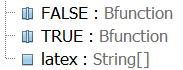
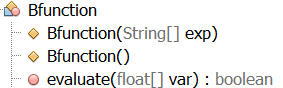




### Bfunction.java

Similar to the idea of Nfunction, in the expression of coordinate patch, we need boolean function expression with respect to coordinates, to express whether a point lies in the effective range of one patch or not, etc.

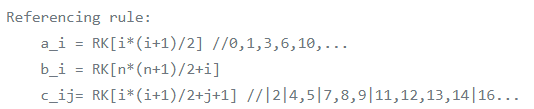
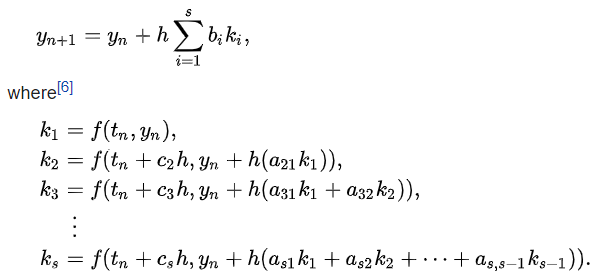
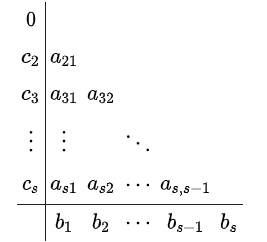
Similar to Nfunction, this just replaced the return value of float in Nfunction to boolean value.

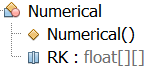


### Numerical.java

This file stores all available Runge-Kutta methods able to be used in this program.

Each method’s Butcher tableau is represented in following way:





## WColor(package)

### WFuntcion.java

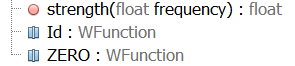
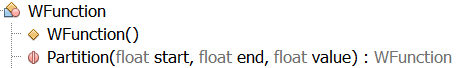
To make a physically viable rendering engine, we can no longer treat light as simply the composition of red green and blue. Instead, we have to analysis what light really are:

The light, from the physics perspective, is a composition of monochromatic light waves of different frequencies(respectively, wavelengths). The light luminosity of a fixed wavelength is a density on the wavelength space. Each wavelength of light corresponds to a RGB value. The RGB configuration we see is just the integration of RGB value measure to the respect luminosity density. But in the low speed Newtonian spacetime point of view, wavelength shifts don’t happen(or at least in a noticeable level), thus there is no need to separate light into different frequencies composition, RGB value is the only observable for human eyes.

However, in relativistic level, even in the simplest case(i.e. Minkowski(flat) spacetime), the doppler shift is already significant, apart from other various effects of curved spacetime. This made us have no other choice other than using the most fundamental way(in geometric optics context) to describe the light.

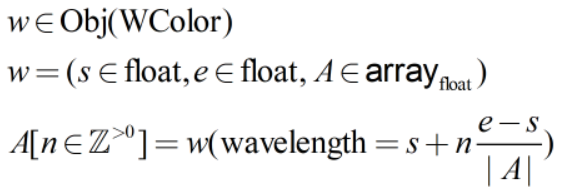
Wavelength function(WFunction) is the first step towards this goal. It is similar to the Nfunction, but there is only one input variable(*float wavelength*), to give the output(*float*, represents the density on that wavelength) using the *strength* method.

*Partition(float start,float end,float value)*: output a WFunction which is only not 0(having constant value of *value*) within the wavelength range [*start*,*end*]



### WColor.java

Following from the upper part of heuristic discussion about light, here is the one-step-further product. WColor is composed of 3 datas: 2 float represents the recorded wavelength section [e,s], and a float array represents the luminosity distribution samples within the range



This is the one class used to record the color obtained from each lightray and used to render the colors on the final screen. Since it only records finite samples of the spectrum, the rest of the spectrum is obtained by linear interpolation between samples. The function *getW* and *addW* all use this interpolation method to modify WColors.

To satisfy the physics laws, there is a few methods used in the program:

*MultiplyW(WFunction g)*: returns the pointwise product with the WFunction g: 

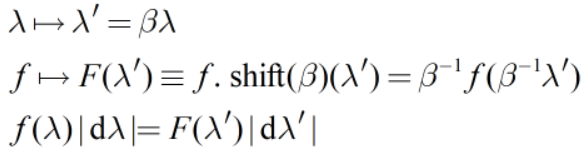
Corresponding method for WColor is *filt(WColor fi)*, used in filtering light that passes through an object.

*addW(WColor)*: returns the result of summing 2 WColor’s spectrum.



Corresponding method for WFunction is *fillAddW(WFunction f)* mostly used when initializing light rays.

*shift(float factor,boolean preserveRange)*: this returns a WColor with spectrum shift by *factor*. The range(i.e. *startW endW*) will correspondingly change with *preserveRange* option being *false*. Notice that WColor is often used in representing light spectrum which is a density(usually regarded as max rank differential forms) on the space of light wavelengths, thus it’s value also has to do a corresponding change.



*mShift* is similar to this method, but without treating the underlying WColor as a density distribution.

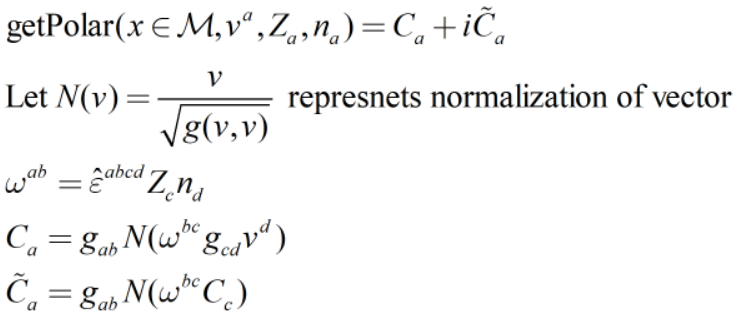
There are also methods that manipulate *WColor* arrays, which is again the tautological extension of *WColor* methods, like *multiply*.

*WColor* arrays usually have an even number of elements, the first half representing the real part of light amplitude, the second half representing the imaginary part of light amplitude. There is a *complexAmplitudeMuti* method, which does a complex number multiplication to the *WColor* array passed in.

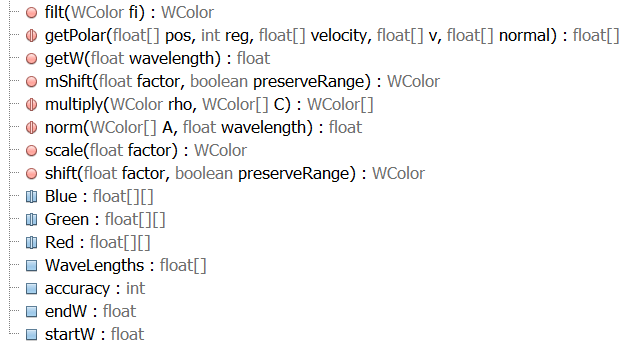
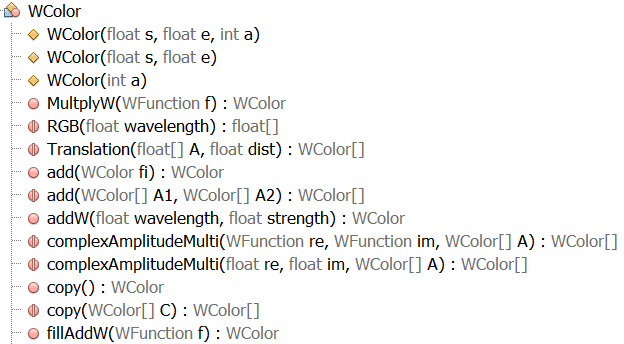
*Translation(float[] A,float dist)*: it gives a *WColor* array where the value of a wavelength satisfy the following:



*getPolar(float[] pos, int region, float[] velocity, float[] ObjectVelocity, float[] ObjectNormal)*: returns a float array(a complex vector) called Amplitude determined using those parameters, using the following formula:



*norm(WColor[] Amplitude,float wavelength)*: returns the *Amplitude*’s density complex vector norm value at *wavelength* inside spectrum space.

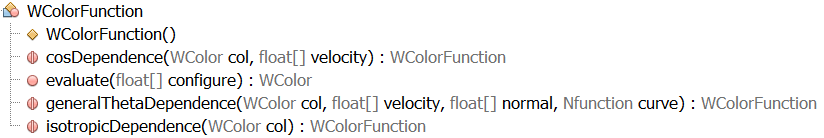


As I finished the first stage of the program, I gradually realized using the WColor array to record each lightray is getting more unnecessary and having too many disadvantages. I will directly record which Object is being hitted instead of WColor in the future development.

### WColorFunction.java

WColorFunction is a “product” of WColor and Nfunction, it gives a WColor with each configuration array passed in using the *evaluate* method. This is used in defineding object’s light emission. This is removed from the *Obj class*.

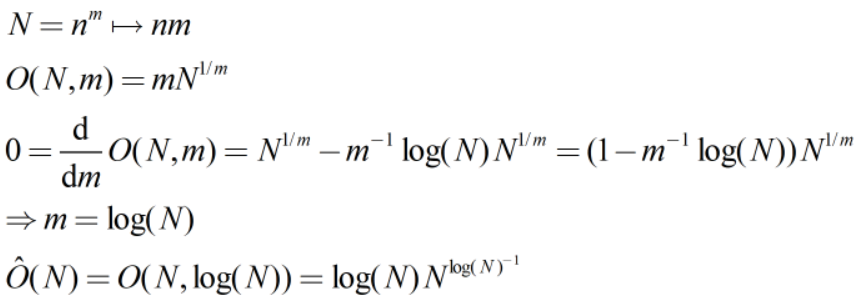
The other static methods in this class are all constructors according to the parameters passed in.

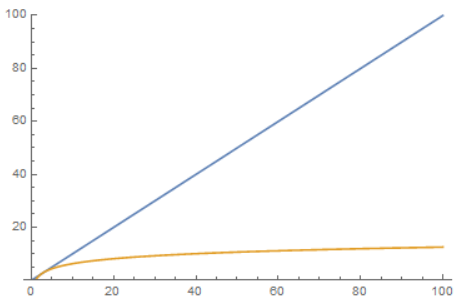


## Format(package)

### ObjFolder.java

The strict law of ray tracing(which uses the framework mentioned at start the document) is that the rendering time is proportional to the number objects, which can be very unfriendly when rendering a large number of objects. That’s why for this program, I came up with the ObjFolder system, which can be used to classify objects. Each Obj (and objFolder) has an envelope cube parameter, which is the cube that completely covers every object inside. The lightray will need to calculate the possible object hitting, only if the lightray was within the envelope cube. Otherwise this will not be needed. With the right classification, the complexity can be reduced from n to the power of m to n times m. the best optimization is shown below, where N is the total number of objects, and m being the number of things inside one folder



Original complexity(blue) and Optimized complexity(yellow)

The ObjFolder is composed of 5 datas:

* subObj(*arrayList<Obj>*), contains all the objects within this folder
* subFolder(*arrayList<ObjFolder>*), contains all the folders within this folder
* previousFolder(*ObjFolder*), pointing to its previous folder. If there is no previous folder, it is assumed to be the “empty” ObjFolder
* envelopeCube(*float[2][4]*), which records the min coordinate vertex and max coordinate vertex of the cube.
* autoUpdate(*boolean*), record whether envelope cube will auto update after objects(or object folders) inside have changed their envelope cube.

and one unique identifier(*int characteristics*) used to distinguish between object

The methods are mostly easy to understand and easy to read if consider ObjFolder as an extension of usual file folders, so I would only introduce methods that are not as obvious as folder:

*getRegion()*: get the folder’s current region(identifier of coordinate patch, see Region.java for explanation)

*reEnvelope()*: recalculate the envelope cube so that it can envelope all of the current objects and folders of objects inside

*within(float[] pos)*: check if the *pos* is inside the envelope cube

**Category of Folder system:**

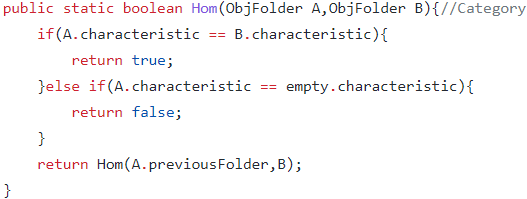
We can define a category on this folder system:

The objects of this category are all folders, and the morphisms between them (A→B) are the relation: B is A(or one of A’s subfolders, or one of one of the A’s subfolders’ subfolders, or …), naming B is within A(B∈A, noticed in this abused notation ∈ does not mean “as an element of”, since we haven’t gave them any set structures)

We can verify the morphisms satisfy the definition of category: A∈A is the identity morphism, and morphisms can be composed(if B∈A(A→B) and C∈B(B→C), then C∈A(A→C=A→B→C))

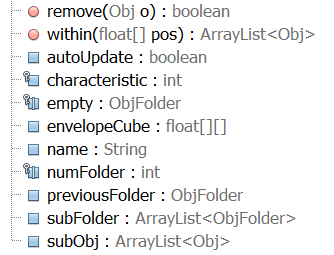
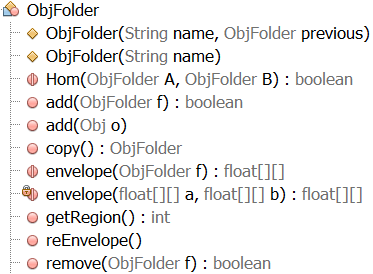
We can further analyze these morphisms: this category is rather simple, all objects form a set, and all morphisms also form a set(which is called a small category). All morphisms that is not identity can be generated from(finitely composed of) morphisms between osculate objects(subfolder relations), we can therefore derive a theorem: A→B if and only if A=B or A.previousFolder→B. Using this, we can wrote a iterative algorithm of Hom functor(a collection of morphisms between 2 designated objects):

*Hom(ObjFolder A, ObjFolder B)*: returns a boolean of weather the hom set(collection of morphisms) between A and B is not empty(exists morphisms between A and B) or being empty



If the hom set is not empty, it might not be unique, since there might be objectFolders which forms a loop, giving infinite amounts of different morphisms:(e.g. B is subfolder of A(refer this morphism as f), C is subfolder of B(refer this as g), and A is subfolder of C(h), then apart from the identity morphism(id\_A:A→A), there is morphism h∘g∘f:A→A(A∈C∈B∈A), refer as l, and l∘l:A→A(A∈C∈B∈A∈C∈B∈A), …), for our folder system, we do not want having loop folders, which can cause unnecessary problems like infinite loops. So we add an extra step of checking in the method *add* to make sure this wouldn’t happen.

With this condition being implied, the initial object in this category does always exist, which is the *empty* objFolder by our initialization of ObjFolders. The terminal object, however, does not general exists(it exists if and only if there is only one branch of folders, and it will be the last folder of the chain)



### Obj.java

Back to our fundamental rendering objects, which is expressed as *class Obj*

*class Obj* has 3 types, which is distinguished using *objType*(*int*)

objType = 0: 3 dimensional simplex. Its *vertices* parameter(float[4][4]) taken the form:

* First vertex position
* Second vertex position
* Third vertex position
* Fourth vertex position

objType = 1: 3 dimensional parallelogram. Its vertices *parameter*(float[4][4]) taken the form:

* First vertex position
* Position of the first vertex that has a direct edge connected to the first vertex
* Position of the second vertex that has a direct edge connected to the first vertex
* Position of the third vertex that has a direct edge connected to the first vertex

objType = 2: 4 dimensional elliptical shape. This is described by a symmetric bilinear form B(x-b,y-b) where the displayed shape is a set where elements x(∈current coordinate patch) inside are inside another designated 4 dimensional parallelogram that has property: B(x-b,x-b)=1. Note that the B need not be positive-definite(which made it able to display not only 3-sphere, but also 3-quadratic surface and 3-hyperbolic surface)

Its vertices *parameter*(float[9][4]) taken the form:

* Display bounding box’s first vertex position
* Display bounding box’s position of the first vertex that has a direct edge connected to the first vertex
* Display bounding box’s position of the second vertex that has a direct edge connected to the first vertex
* Display bounding box’s position of the third vertex that has a direct edge connected to the first vertex
* Display bounding box’s position of the fourth vertex that has a direct edge connected to the first vertex
* The symmetric bilinear form’s first line of component
* The symmetric bilinear form’s second line of component
* The symmetric bilinear form’s third line of component
* The symmetric bilinear form’s fourth line of component
* The coordinate of the symmetric bilinear form’s barycenter

I will use to represent the *i* th vertex in the following text

Interchangeable parameters:

* *reflection*(*float*): possibility of light get reflected
* *transparency*(*float*): possibility of light passing through the object
  + The sum of the 2 above should be less than1. The rest of the probability represents the rate of light getting emitted from the object.
* *name*(*string*): name of the object
* *filter*(*WColor*): the spectrum filter of this object. Usually less or equal to 1 everywhere, representing some of the light getting absorbed. But it could also be greater than 1, representing the part of the light spectrum that has been amplified.
* *emittion*(*WColor*): a typo here(emission), represents the spectrum of the light that emitted from the object.
* *normals*(*float[][]*): not a required parameter(auto filled once the vertices has been filled/modified), records the normal vector of each surface of the object
* *velocity*(*float[]*): not a required parameter. the velocity of this object, a co-vector(dual tangent vector, this way its component wouldn’t change(with only a possible rescaling) with coordinate(metric)), it will auto filled by the first normal vector, which is used in calculating spectrum emission(doppler shift, etc.)
* *previousFolder*(*ObjFolder*): records its previous object folder
* *protected characteristic*(*int*): the unique identifier of the object

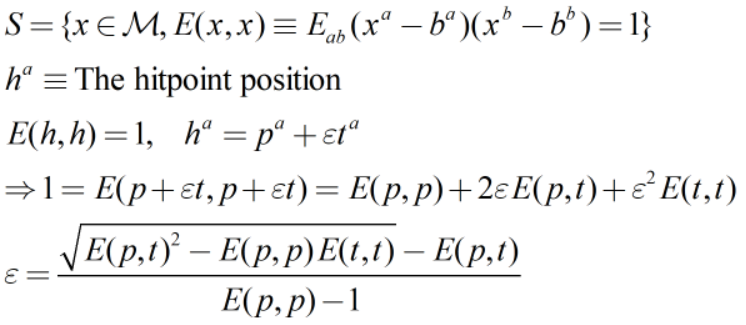
*envelope(Obj a)*: gives the envelope cube parameter(*float[2][4]*) for the object a, by complete all the vertices of the object using given object’s type and vertices, and pick min & max over all vertices, with a range of expansion(*float expandEnvelope*) being added which is used to avoid *stepSize* being to big that overrides the object itself(thus is needed to be bigger than *stepSize*).

*checkHit(float[] pos,float[] tan)*: returns the lightray’s estimate hitting scenario in *float[5]{*hitPosT,hitPosX,hitPosY,hitPosZ,ε*}* with given parameter(*pos*: ray’s last iteration position, *tan*: ray’s current position - last iteration position, as a difference in light ray’s coordinate, coincides with velocity in flat spacetime). ε represents satisfies the equation: pos + ε tan = hitpos

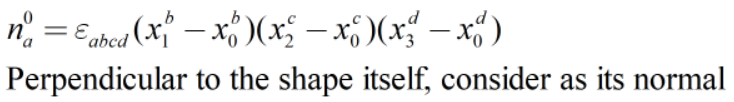
For 3-simplex and 3-parallelogram:

* To actually know whether the lightray passes through the object, one first considers whether the lightray’s most recent trajectory(coordinate based) intersects with the plane the object is in. I used the equationto determine the value of ε, which is how far the extension line(coordinate based, starts from pos(*p*), with tangent vector tan(*t*)) will intersect with the plane.
* If ε’s value is within the interval [0,1], represents it does hit the plane, then we further consider if the ray is outside of the shape itself or not using all other “normal” vectors of all other the surface with similar strategy: (no summation of index *i* here)to see if the hitpoint is within the shape
* If it satisfies all the conditions above, return the hit point pos and the value of ε, otherwise returns {0,0,0,0,-1}

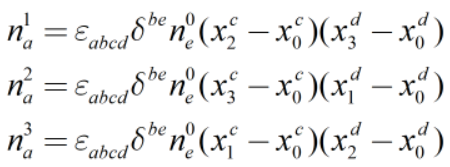
For elliptical shape, the process is bit more complex:

* It uses the elliptical bilinear form to work out the hit position to the elliptical shape:
  + 
  + It is already suffice to check the positive root, since the negative root represents the point in elliptical boundary which the ray is already passed, and imaginary value will arise NaN means no chance to hit it
* It then check if the hit point is within the bounding box with similar process for all 8 3-surfaces of the 4-parallelogram: (again, no summation to *μ* applies here)
* If it satisfies all the conditions above, return the hit point pos and the value of ε, otherwise returns {0,0,0,0,-1}

*setVertices(float[][] vert)*: is responsible for vertices resetting and recalculate normal vectors with the formula:



Interchangeable normals for 3-simplex and 3-parallelogram:

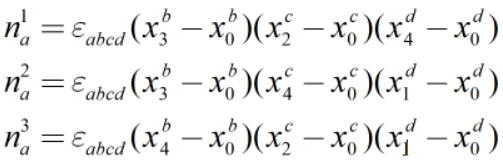


Special for 3-simplex(since other 3 2-surface of the 3-parallelogram is just opposite of these 3)



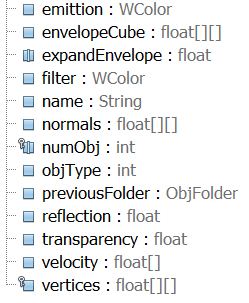
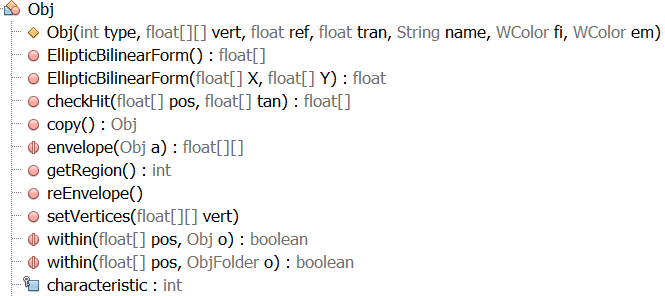
Notice that these normals are coordinate based(not just up to a rescaling) since one uses the delta(identity matrix in the current coordinate), but it does not affect the intersection calculation, as long as the metric is positively-definite, the result would not change.

Special for 3-elliptical surface bounding box:



The 3 normals here are not coordinate dependent(up to a rescaling)

The rank of the product normals and vertices has been specially chosen so that the orientation of the shape would not change the algorithm result.



### Region.java

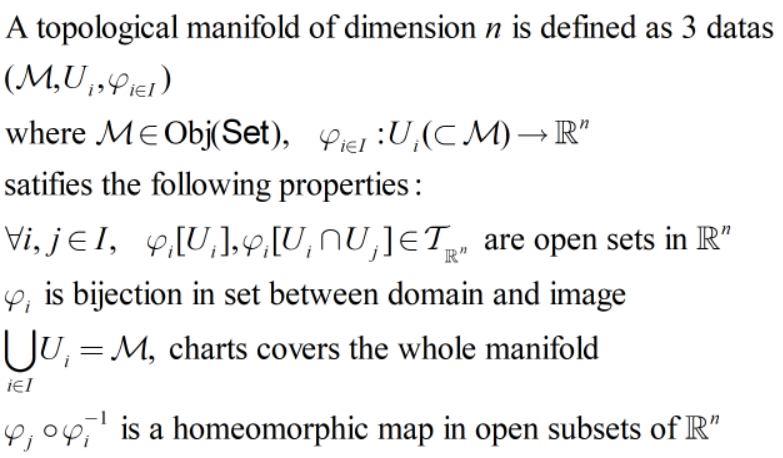
Finally the part that relates to the underlying background of spacetime: A (pseudo-)Riemannian manifold.

First we shall introduce what is a manifold. Bluntly speaking, it is just a fancy word of a surface(usually smooth, attribute word of differentiability is added when it is a manifold with such differentiability) with consistent arbitrary dimension over the whole space.

Examples include all kinds of nice looking surface one can think of, such as Euclidean space, open subsets of the Euclidean space(e.g. open disks and open balls), n dimensional sphere, n-dimensional torus, cylinders, any riemann surface with poles cutted, projective spaces, any (finitely cartesian)product spaces of the aboves, etc.

But how exactly are we going to rigorously define them in terms of mathematics instead of unspeakable intuitions? Of all the examples above, one can see that despite they might all have obviously different global properties(topological properties like number of n-holes), when looking at them in local views, these surfaces are smooth, simple just like our familiar Euclidean space. This gives the definition of topological manifold:

Using charts to make each local part of the manifold looks like Euclidean space



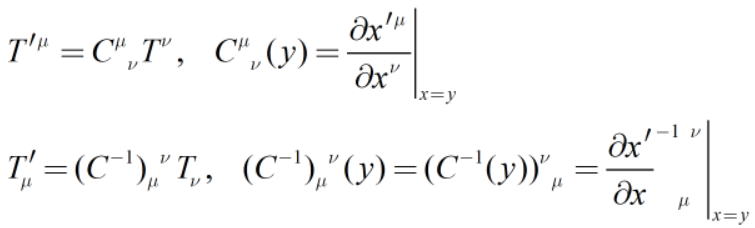
The final condition make sure in the overlapping areas between charts, the notion of topology(open sets) is consistent, thus induced a topology on the manifold, inherits most of the merit of Euclidean space such as usual notion of open intervals(balls), usual notion of limit and convergence, points being separable(Hausdorff property), etc.

To make it a manifold, one adds the compatibility condition: 

This way, the naturally induced differentiating notion of charts will not conflict with each other. The charts here also pullbacks the natural coordinate in Euclidean space, thus giving a local coordinate, which is said to be a coordinate.

To make it has a nice notion of measurement(which gives the notion of curve length(proper time), tangent vector inner product, translation, length and time(in spacetime), etc.), one adds the metric to it:(a smooth symmetric bilinear form) The manifold is called Riemannian when the metric at each point is positively-definite, pseudo-Riemannian if it is non-degenerate, and Lorentzian if its maximum dimension of negatively-definite subspace is 1, which is also called a spacetime.

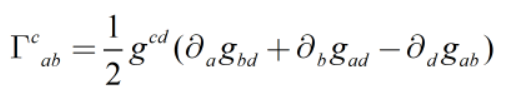
In this program, I wrote the class Region to realize the notion of manifold:

Each coordinate(patch) are represented with an instance of Region, which records the relation between nearby coordinates(*int[] adjacent*, which records all the adjacent coordinate patches’s code(unique identifier)) in overlapping area with *Ntensor[] coodTran*, expressed in {four component tensor represents how each new coordinate varies with old coordinate, <=, …}, and automatically calculated coordinate transformation matrix for tangent spaces(*Ntensor[] coodTranMat*) and dual tangent spaces(*Ntensor[] coodTranMatInv*) with formula: 

This can also be manually filled within *Ntensor* syntax to (possibly) reduce the computation complexity, thus increasing the efficiency of the program, the same time improving the accuracy.

The Lorentzian metric *Ntensor gab* is represented with a 16 component symmetric Lorentzian *Ntensor*. Its inverse metric is *Ntensor gAB*, also being auto calculated using pointwise *Tensor* method *Adjugate4* and *determinant4*, manually filled is also recommended with reasons above.

Same thing applies to *Gamma*, which is a 64 component *Ntensor*, called the Levi-Civita connection, a unique torsion free metric compatible connection on the frame bundle, auto calculated using the following formula:



Manually filling is also recommended.

There are also parameters *String[] coodSymbol* used to tell the program each coordinate symbol such as default {t,x,y,z}, in latex syntax.

*String name* records the wanted display name of the region

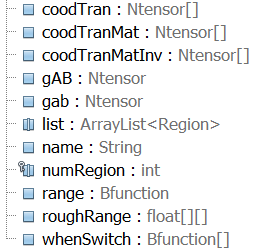
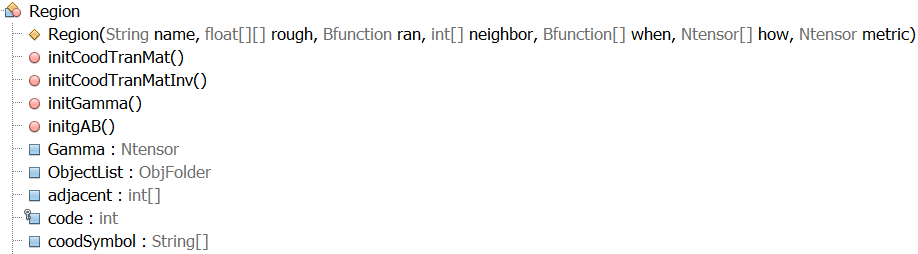
*float[2][4] roughRange* gives the envelope cube of the effective range of the coordinate

*Bfunction range* is a boolean valued function based on coordinate, gives whether the coordinate is still effective(within Ui in the manifold definition) at the passed in position(in coordinate)

*Bfunction[] whenSwitch* gives valid adjacent coordinate patches, used when outside of current coordinate’s effective range and trying to switch the coordinate.

*ObjFolder* ObjectList is a ObjFolder with previousFolder *empty* that contains all the objects inside this coordinate patch.

*static arrayList<Region> list* gives all the Regions created in the program so far, making easy access to related datas such as usually seen reference: “*Region.list.get(region).<region parameters>*”



## GUI(package)

### GUI.java

*class GUI* is a class used in *class Core* to showing the rendering result(also the progress bar which is currently not available due to unknown reason)

It has 2 separate mode:

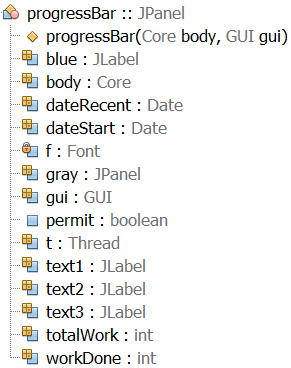
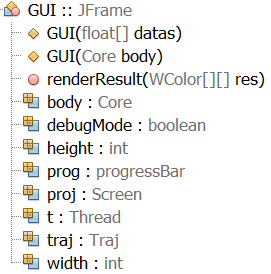
Debug mode, which created using *GUI(float[] datas)*, it will display the trajectory of geodesics(recorded using *datas* with the format {t0,x0,y0,z0,t1,x1,y1,z1,t2,x2…})

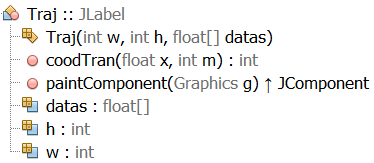
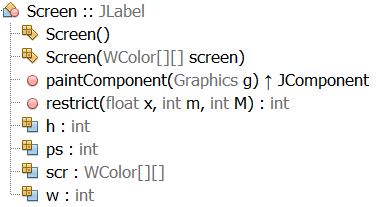
And render mode, which is created using *GUI(Core body)*, passing the current *Core* instance which is responsible for calculation.

*class Traj* is an extension of *JLabel* GUI component, which paints trajectory from *datas* in the debug mode, with passed in parameter *int w*(screen width(#pixels)), parameter *int h*(screen height(#pixels)) and *float[] datas*. Currently hard coded to only render x-z plane, but would be able to adjust in the future plan.

*class Screen* is also an extension of *JLabel* GUI component, which paints the rendering result from *WColor[][] screen*.

*class progressBar* is an extension of the *JPanel* GUI component, ideally would display a blue bar at the center of screen showing how much work is done, estimate time left and time used with obvious implementation. It is worth noting that the *boolean* parameter *permit* is used to stop the program since it is a separately running thread that constantly checks the work done every 0.5 second(ideally). For some unknown reason, it is not able to show on the screen.





## Engine(package)

(abandoned since GPU is no longer supported on my device)

# Appendix A: Source Code

Source Code(Jan 31, 2022): <https://github.com/mcxzx/GeneralRelativity-Visual-Project>

# Appendix B: Alpha/Beta Testing

| **Requirement Spec**  **Reference** | **Software Feature(s)** | **Code Block(s) Tested** | **Status**  (Pass/Fail or  Pass/Fail with explanation) | **Date Tested** |
| --- | --- | --- | --- | --- |
| 4.1.1.1 | Correctly progressed tensor multiplication and number multiplications | class Tensor  scale, product, producti | Passed | Nov. 1 |
| 4.1.1.2 | Correctly progressed tensor product and their contractions | ||  product | Passed | Nov. 1 |
| 4.1.1.3 | Correctly progressed tensor transpose | ||  transpose | Passed | Nov. 5 |
| 4.1.1.4 | Correctly outputs identity matrix and Levi-Civita Symbol | ||  IdentityMatrix, LeviCivitaSymbol, I4, LC3, LC4 | Passed | Nov. 15 |
| 4.1.1.5 | Correctly progressed matrix inverse and determinant | ||  Adjugate4, determinant | Passed | Jan. 7 |
| 4.1.1.6 | Valid special support for 4d spacetime tensor operations that is allowed in GPU processing | ||  All kinds of method ended with 4 | Passed with modification since the gpu was not available, and passed | Nov. 15 to Jan. 25 |
| 4.1.2.1 | Accuracy-customizable representation of compositions of wavelengths | class WColor/WFunction | Passed | Jan. 26 |
| 4.1.2.2 | Valid add & subtract operations between WColors | ||  Add, fillAddW | Passed | Jan. 26 |
| 4.1.2.3 | Valid number multiplication and scalar field action on WColors | ||  MultiplyW, filt, scale | Passed | Jan. 26 |
| 4.1.2.4 | Valid Redshift & Blueshift operation of WColors | ||  mShift, shift | Passed | Jan. 26 |
| 4.1.2.5 | Valid conversion from Wavelength functions to WColors | ||  fillAddW from new instance | Passed | Jan. 26 |
| 4.1.2.6 | Valid conversion from black body radiation to WColors |  | Not yet implemented |  |
| 4.1.2.7 | Valid conversion from WColors to RGB colors | ||  RGB | Passed | Jan. 26 |
| 4.1.3.1 | Able to run the program on supported GPUs or CPUs |  | Expectation removed since GPU is no longer supported |  |
| 4.2.1.1 | Distribute each region with a natural number index | Region  code | Passed | Nov. 17 |
| 4.2.1.2 | Record a rough 4-cube that includes the whole region. Allows infinity in certain directions. | ||  roughRange | Passed | Jan .24 |
| 4.2.1.3 | Record a simple and specific boolean expression in coordinate claims the effective covering region subset of the 4-cube | ||  range | Passed | Jan. 24 |
| 4.2.1.4 | Record the way it displays in 3d euclidean space when not rendering |  | Not yet implemented |  |
| 4.2.1.5 | Record the WColor image displayed when the light ray iteration ends(the default should be black) | GUI, Screen, Core  screen | Passed | Jan. 26 |
| 4.2.1.6 | Record an expression of the metric in the region | Region  gab | Passed | Dec. 17 |
| 4.2.1.7\* | Record expressions of the inverse metric, first derivative of the metric and the Riemann Curvature Tensor in the region | ||  gAB | Passed with modification: Riemann curvature tensor expectation removed since it is not necessary | Dec. 17 |
| 4.2.2.1 | Record expressions of the attached region index and corresponding new coordinate in terms of the old coordinate | ||  Adjacent, coodTran, coodTranMat, coodTranMatInv | Passed | Dec. 17 |
| 4.2.2.2 | Set attached region index negative if it is the edge of the spacetime |  | Passed with modification(not actually used in the program) | Jan. 26 |
| 4.2.3.1 | Record the shape of the object surface in the coordinate system(3-cube, 3-sphere or 3-simplex) | Obj  objType | Passed | Nov. 16 |
| 4.2.3.2 | Record the reflection rate of the surface | ||  reflection | Passed | Nov. 17, Jan. 26 |
| 4.2.3.3 | Record the WColor of the light it emit on the surface | ||  emittion | Passed | Nov. 17, Jan. 26 |
| 4.2.3.4 | Record the WColor filter of the surface | ||  filter | Passed | Nov. 17, Jan. 26 |
| 4.2.4.1\* | Record the expression of precalculatable light propagation |  | Expectation removed |  |
| 4.2.4.2\* | Record the killing vector field at each point in its component form |  | Expectation removed |  |
| 4.2.4.3\* | Record the special geodesics displayed, its RGB color | GUI, Traj  datas | Passed before. Not checked after updating | Jan. 14 |
| 4.3.1.1 | isLightRaySplitting & maximum #splitting rays |  | Expectation removed |  |
| 4.3.1.2 | Stepsize and the maximum #step | Core, RayCalc  stepSize, maxStep | Passed | Jan. 26 |
| 4.3.1.3 | Resolution of the rendering | Core, RayCalc  scrWidth, scrHeight, RayCalc | Passed | Jan. 26 |
| 4.3.1.4 | Light ray WColor accuracy | WColor  accuracy | Passed | Jan. 26 |
| 4.3.1.5 | Method of solving PDE(rank n RK method or adaptive RK method) | RayCalc  run | Passed | Jan. 26 |
| 4.3.1.6 | Rendering platform(GPUs, CPUs, Java Alternative Algorithm or Java Thread Pool) |  | Passed with modification, GPU no longer supported | Jan. 26 |
| 4.3.2.1 | Record the location of the camera(region index, coordinate) | Core, RayCalc  camCood | Passed | Jan. 26 |
| 4.3.2.2 | Record the 4-frame of the camera(need to be all orthonormal) | ||  camFrame | Passed | Jan.26 |
| 4.3.3.1 | Basic parameters: WColor, filters, coordinate, region index, wave-vector, stepsize, #steps, strength, Frame Tensor | Obj, Region, RayCalc  filters, stepSize, maxStep | Passed with modification: Frame Tensor removed, for the instability of evolution equation and efficiency consideration | Jan. 26 |
| 4.3.3.2 | Iteration evolution obeys the evolution PDEs: | RayCalc  run | Passed | Jan. 26 |
| 4.3.3.3 | Iteration evolution shall auto change region index and coordinate when evoluted to the current region edge, and transform it’s tensor value data components. |  | Not yet implemented |  |
| 4.4.1.1 | Angle of visual in terms of 4-frame in a lorentzian vector space |  | Not yet implemented |  |
| 4.4.1.2 | is glare effect and diffraction effect enabled |  | Not yet implemented & modified |  |
| 4.4.1.3 | Output display resolution | GUI, Screen | Passed |  |
| 4.5.1.1 | Consists of buttons named “Settings”, “File”, “Rendering”, “Help”, “About” |  | Not yet implemented |  |
| 4.5.1.2 | “Settings” menu shall contain options “Tab settings”, “Text settings”, “Preference”, “Languages” |  | Not yet implemented |  |
| 4.5.1.3 | “File” menu shall contain options “New project”, “Open project”, “Save project”, “File information”, “Export project”, “Import project” |  | Not yet implemented |  |
| 4.5.1.4 | “Rendering” menu shall contain options “Start new rendering”, “Pause rendering”, “Continue rendering”, “Abort rendering”, “Export image as” |  | Not yet implemented |  |
| 4.5.1.5 | “Help” menu shall contain options “User guide”, “Instruction manual”, “Working mechanics”, “(End)Help mode” |  | Not yet implemented |  |
| 4.5.1.6 | “About” shall link to the info & copyright of this software |  | Not yet implemented |  |
| 4.5.2.1 | It shall has a sidebar on the right which is able to expand/close this tab |  | Not yet implemented |  |
| 4.5.2.2 | This tab shall contain basic info of current spacetime project, and options “Spacetime manifold”, “Objects”, “Camera”, “Rendering”, “Special functionalities”, “Captures” |  | Not yet implemented |  |
| 4.5.3.1 | It shall has a sidebar on the left which is able to expand/close this tab |  | Not yet implemented |  |
| 4.5.3.2 | This tab shall display options to adjust various parameters of selected topic on the left tab |  | Not yet implemented |  |
| 4.5.3.3 | The camera parameters should be always on the top able to adjust using value typing, dragging sliders, and keyboard input(WASD, space, shift, numpad or arrow keys) |  | Not yet implemented |  |
| 4.5.3.4 | If select “Rendering” on the left, it shall display parameters in “4.3.1” and “4.4.1” |  | Not yet implemented |  |
| 4.5.3.5 | If select “Spacetime manifold” or “Objects” on the left, see “4.5.5” and “4.5.6” for details |  | Not yet implemented |  |
| 4.5.3.6 | If select “Captures” on the left, it shall display all records of rendering. Click on one would show that on the background screen |  | Not yet implemented |  |
| 4.5.4.1.1 | It shall display every objects with its normal RGB color in the current region with the rules of this coordinate system |  | Not yet implemented |  |
| 4.5.4.1.2 | It shall also display a rough region of overlapping area |  | Not yet implemented |  |
| 4.5.4.1.3 | It shall display the normal RGB image of “4.2.1.5” as a box in the background |  | Not yet implemented |  |
| 4.5.4.1.4 | It shall react quickly as the parameters of the cam changes |  | Not yet implemented |  |
| 4.5.4.1.5 | It shall be able to view separated from the dedicated rendering cam |  | Not yet implemented |  |
| 4.5.4.2.1 | Keyboard input for cam moving should be blocked for now |  | Not yet implemented |  |
| 4.5.4.2.2 | A gray translucent overlay shall be above the “non-rendering view”, with a progress bar and estimated evolution time. |  | Not yet implemented |  |
| 4.5.4.3.1 | A panoramic picture shall be displayed on this screen with given rendering parameters |  | Not yet implemented |  |
| 4.5.4.3.2 | The view of the panoramic picture shall be able to drag by mouse, using keyboard numpad(or arrow keys), or adjust cam parameter on the right tab |  | Not yet implemented |  |
| 4.5.4.3.3 | The picture should be able to react rather quickly with changing of post-rendering parameters |  | Not yet implemented |  |
| 4.5.4.3.4 | Right tab should display a big button “back” to return to the state of “4.5.4.1” |  | Not yet implemented |  |
| 4.5.5.1 | “Add new region”, “modify current region” options and the list of all regions shall displayed on the right tab |  | Not yet implemented |  |
| 4.5.5.2 | The rough region should be display on non-rendering background screen if it is overlapped with current region |  | Not yet implemented |  |
| 4.5.5.3 | Clicking any region button inside the list will unfold a tab which includes more info of this region, such as expressions of metric, overlapping expressions, etc. |  | Not yet implemented |  |
| 4.5.6.1 | “Add new object” option and the list of all objects in this region shall be displayed on the right tab |  | Not yet implemented |  |
| 4.5.6.2 | Once clicked “add object”, a tab will show up on the top of the right tap, includes the basic parameters of the object, see “4.2.3” for details |  | Not yet implemented |  |
| 4.5.6.3 | Keyboard input from numpad(arrow keys or mouse dragging) and mousewheel(key []) shall be used to determine the placement of the object |  | Not yet implemented |  |
| 4.5.6.4 | Clicking any object in the list on the right tab shall unfold a tab showing all basic info of the object |  | Not yet implemented |  |