CS:AI - Language Engineering
3D GeoServer

Arsen Bilyalov, Steven Huygens

25 May 2024

1 Introduction

The 3D GeoServer in essence keeps track of objects in a three-dimensional space. Utilising the principles of functional programming in Haskell, these primitive objects can be composed into more complex regions. The regions have been implemented in a deep-embedded way, which allows to easily write different interpretations for these regions. To illustrate this further we have implemented three interpretations: InRegion, SerializeRegion and RenderRegion. Each interpretation makes use of a recursive scheme to traverse the abstract syntax tree and process each region constructs.

2 Deep embedding

Deep embedding in Haskell involves representing a domain-specific language within Haskell by defining its syntax explicitly as data types. This approach allows for the creation of abstract syntax trees (AST) that can be manipulated, analyzed, and interpreted within the host language. With deep embedding, each construct of the DSL is mapped to a corresponding data type in Haskell, providing a clear and flexible structure that facilitates various operations such as optimization, transformation, and evaluation. This technique is powerful for implementing complex languages and tools, as it leverages Haskell's strong typing and functional programming paradigms to create robust and maintainable code.

To embed these regions we have defined a data type `Region` with the AST illustrated in listing 1. Each constructor defines a specific way to compile a node in the AST. The interpretations of regions are implemented using an algebra. To allow for the flexibility of defining new interpretations, a data type `RegionAlgebra` is necessary (listing 1). `RegionAlgebra` has a parametrized type `a` which depends on the interpretation. For example in the case of serializing regions `a` is a `String`. `RegionAlgebra` is explained in more detail in section 4

Some of the Region constructors have one or more Regions as parameter which enables to compose more complex regions consisting of cubes, spheres and cones. For this we need a recursive scheme that is able to parse this tree. This is defined in `foldRegion`. `foldRegion` applies a particular `RegionAlgebr`a recursively on the AST. The implementation of `foldRegion` is shown in listing 2.

```haskell
type RadiusGS = Double
type SideGS = Double
type BaseGS = Double
type HeightGS = Double
type PointGS = (Double, Double, Double)

data Region = -- Abstract Syntax Tree
    SphereGS RadiusGS
    | CubeGS SideGS
    | ConeGS BaseGS HeightGS
    | TranslateGS Region PointGS
    | RotateGS Region PointGS
    | OutsideGS Region
    | IntersectionGS Region Region
    | UnionGS Region Region

data RegionAlgebra a =
  RAlg {
    ra_sphere  :: RadiusGS -> a,
    ra_cube :: SideGS -> a,
    ra_cone :: BaseGS -> HeightGS -> a,
    ra_translate  :: a -> PointGS -> a,
    ra_rotate :: a -> PointGS -> a,
    ra_outside  :: a -> a,
    ra_intersection  :: a -> a -> a,
    ra_union :: a -> a -> a
  }
```

Listing 1: Data type definitions for Region.

```haskell
foldRegion :: RegionAlgebra a -> Region -> a
foldRegion alg (SphereGS r) = ra_sphere alg r
foldRegion alg (CubeGS s) = ra_cube alg s
foldRegion alg (ConeGS b h) = ra_cone alg b h
foldRegion alg (TranslateGS r p) = ra_translate alg (foldRegion alg r) p
foldRegion alg (OutsideGS r) = ra_outside alg (foldRegion alg r)
foldRegion alg (IntersectionGS r1 r2) = ra_intersection alg (foldRegion alg r1) (foldRegion alg r2)
foldRegion alg (RotateGS r p) = ra_rotate alg (foldRegion alg r) p
foldRegion alg (UnionGS r1 r2) = ra_union alg (foldRegion alg r1) (foldRegion alg r2)
```

Listing 2: The recursive scheme defined in foldRegion.

3 Architecture

To ensure more readability and modularity we've split the project into three files. An overview is shown in figure 1All the high level (data) types are defined in Region. RenderRegions provides helper functions to draw the regions

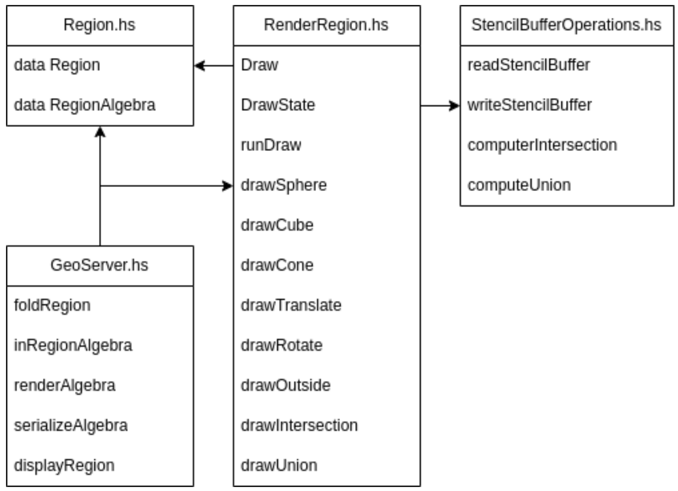


Figure 1: The architecture of the 3D GeoServer. The arrows indicate dependencies. For example renderAlgebra depends on RegionAlgebra and Draw

using the OpenGL and GLUT wrappers in Haskell. The implementation for drawing the intersection, outside and union required tricky calculations to be done on the stencilbuffer. For this reason we've put some helper functions in a seprate file, StencilBufferOperations. The idea is to abstract away the low level programming of OpenGL and provide user friendly functions to the GeoServer. The GeoServer is responsible for the recursion scheme defined in `foldRegion` and implementing the different region algebras. It also contains `displayRegion` which creates a window and renders a region using the `runDraw` function of RenderRegion.

4 Algebras

Now that we have defined different regions we can start using these regions. There are different ways you can interpret these regions. To be able to reuse the defined regions and have different interpretations of these regions we have defined different algebras. We will discuss three different interpretations with their respective algebras. The first InRegion algebra can be used to find out if a given point is inside a region. This algebra is described in Section 4.1. The next algebra can be used to get a string representing the region. This algebra is described in Section 4.2. The last algebra renders the region in 3D. This algebra is described in Section 4.3.

4.1 InRegion Algebra

As discussed before the data type `RegionAlgebra` has a parametrized type `a` which depends on the specific implementation. In this case we want to have a function that for a given point can return a boolean representing whether or not the point lies within the region. For this reason parameter `a` becomes `PointGS -> Bool`. The implementation is shown in listing 3. An example illustrates this in listing 4.

```
inRegionAlgebra :: RegionAlgebra (PointGS -> Bool)
inRegionAlgebra = RAlg {
  ra_sphere = \r (x, y, z) -> x^2 + y^2 + z^2 <= r^2,
  ra_cube = \s (x, y, z) -> abs x <= s/2 && abs y <= s/2 && abs z <= s/2,
  ra_cone = \b h (x, y, z) -> let d = distanceFromCenterConeYAxis b h y
                                in abs x <= d && abs x >= b && abs z <= d && abs z >= b,
  ra_translate = \r (x0, y0, z0) (x, y , z) -> r (x - x0, y - y0, z - z0),
  ra_rotate = \r (a, b, c) (x, y, z) -> r (inverseRotatePoint (a, b, c) (x, y, z)),
  ra_outside = \r p -> not (r p),
  ra_intersection = \r1 r2 p -> r1 p && r2 p,
  ra_union = \r1 r2 p -> r1 p || r2 p
}
```

Listing 3: Implementation of inRegionAlgebra.

```
ghci> sphere = SphereGS 0.5
ghci> cube = CubeGS 0.2
ghci> outsideCube = OutsideGS cube
ghci> intersection = IntersectionGS outsideCube sphere
ghci> foldRegion inRegionAlgebra intersection (0,0,0)
False
ghci> foldRegion inRegionAlgebra intersection (0.3, 0, 0)
True
```

Listing 4: An example of the inRegion algebra.

4.2 Serialize Algebra

This algebra can be used to get the serialization of a region. The implementation of this algebra can be seen in Listing 5. If you use this algebra on a region you will get a string representing this region. This string can be used again to construct the same region. A simple example of this algebra can be seen in Listing 6.

```
serializeAlgebra :: RegionAlgebra String
serializeAlgebra = RAlg {
  ra_sphere = \r -> "SphereGS " ++ show r,
  ra_cube = \s -> "CubeGS " ++ show s,
  ra_cone = \b h -> "ConeGS " ++ show b ++ " " ++ show h,
  ra_translate = \r p -> "TranslateGS (" ++ r ++ ") " ++ show p,
  ra_rotate = \r p -> "RotateGS (" ++ r ++ ") " ++ show p,
```

```
  ra_outside = \r -> "OutsideGS (" ++ r ++ ")",
  ra_intersection = \r1 r2 -> "IntersectionGS (" ++ r1 ++ ") (" ++
    r2 ++ ")",
  ra_union = \r1 r2 -> "UnionGS (" ++ r1 ++ ") (" ++ r2 ++ ")"
}
```

Listing 5: The implementation of the serialize algebra.

```
ghci> sphere = SphereGS 0.5
ghci> cube = CubeGS 1
ghci> translatedCube = TranslateGS cube (0.1, 0.2, 0.3)
ghci> unionCubeSphere = UnionGS sphere translatedCube
ghci> foldRegion serializeAlgebra unionCubeSphere
"UnionGS (SphereGS 0.5) (TranslateGS (CubeGS 1.0) (0.1,0.2,0.3))"
```

Listing 6: An example of the serialize algebra.

4.3 Render Algebra

This interpretation visualizes regions in three dimensions. To implement this we used the OpenGL [2] and GLUT [1] libraries. The Regions consist of three primitive objects: a sphere, a cube and a cone. How to render these objects is discussed in Section 4.3.1. You can apply transformations on these objects: `TranslateGS` and `RotateGS`. How to render objects after these transformations is discussed in Section 4.3.2. If you want to render multiple objects at the same time you can use `UnionGS`. How to render the union of regions is discussed in Section 4.3.5. All the mentioned regions until now can be rendered in three dimensions. However, the last two regions, `OutsideGS` and `IntersectionGS`, can only be correctly rendered in two dimensions. This has to do with the fact that both of these type of regions need to use a stencil buffer to render the regions correctly. How the outside of a region and the intersection of regions is rendered is discussed in Section 4.3.4 and Section 4.3.3 respectively.

To render a region you can use the `displayRegion` function. The implementation of this function and the `renderAlgebra` can be seen in Listing 7.

```
renderAlgebra :: RegionAlgebra (Draw())
renderAlgebra = RAlg {
  ra_sphere = drawSphere,
  ra_cube = drawCube,
  ra_cone = drawCone,
  ra_translate = drawTranslate,
  ra_rotate = drawRotate,
  ra_outside = drawOutside,
  ra_intersection = drawIntersection,
  ra_union = drawUnion
}

runDraw :: DrawState -> Draw a -> IO a
runDraw initialState drawAction = evalStateT drawAction
    initialState

drawing :: Region -> DisplayCallback
```

```haskell
drawing region = do
    clear [ ColorBuffer, DepthBuffer, StencilBuffer]
    -- This is needed for intersections
    stencilTest $= Enabled
    clear [StencilBuffer]
    colorMask $= Color4 Disabled Disabled Disabled Disabled
    stencilFunc $= (Always, 1, 0xFF)
    stencilOp $= (OpReplace, OpReplace, OpReplace)
    drawOverWholeStencilBuffer
    colorMask $= Color4 Enabled Enabled Enabled Enabled
    stencilTest $= Enabled
    -- until here
    runDraw (DrawState (Color3 1 0 0) (Color3 0.8 0 0)) (foldRegion renderAlgebra region)
    flush

displayRegion :: Region -> IO ()
displayRegion region = do
  (_progName, _args) <- getArgsAndInitialize
  _window <- createWindow "GeoServer"
  displayCallback $= drawing region
  mainLoop
```

Listing 7: The renderAlgebra and the main function to render a region.

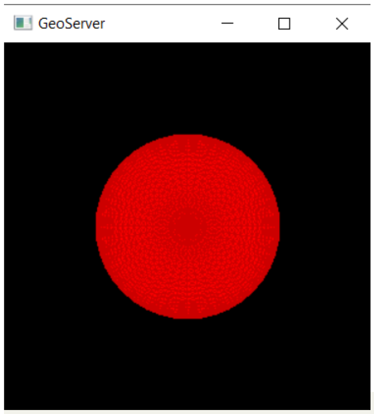
4.3.1 Render Basic Objects

Using OpenGL and GLUT it becomes straightforward how to render spheres, cubes and cones. The code to render these objects can be found in Listing 8. Examples of this code being used to render objects can be found in Section 9.

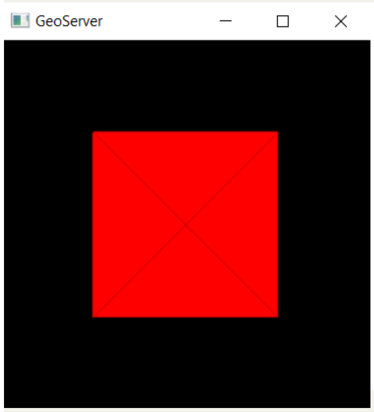
```haskell
drawCone :: GLdouble -> GLdouble -> Draw()
drawCone base height = drawWithOutline $ renderObject Solid (Cone base height 40 40)

drawSphere :: GLdouble -> Draw()
drawSphere radius =  drawWithOutline $ renderObject Solid (Sphere' radius 40 40)

drawCube :: GLdouble -> Draw()
drawCube side = drawWithOutline $ renderObject Solid (Cube side)
```

Listing 8: The code used to render spheres cubes and cones.

```
ghci> s = SphereGS 0.5
ghci> displayRegion s
```




```
ghci> c = CubeGS 1
ghci> displayRegion c
```

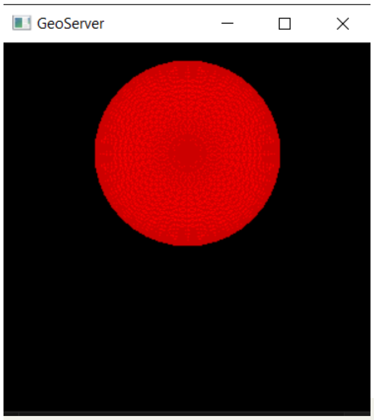


Listing 9: Examples of regions being rendered.

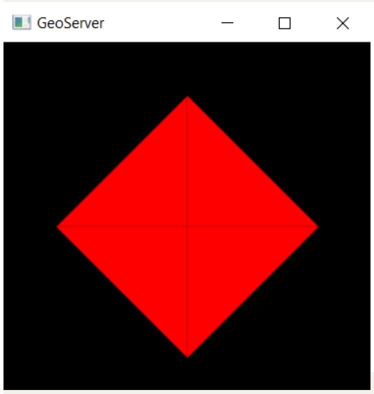
4.3.2 Render Transformations

Translating and Rotating a render in OpenGL is also straightforward. This can be done by using the `preservingMatrix` and using the `translate` and `rotate` functions of OpenGL. Then you can render the regions that had to be translated or rotated. An example of a sphere being translated in the positive y-direction and an example of a cube being rotated 45° around the z-axis can be seen in Listing 10.

```
ghci> s = SphereGS 0.5
ghci> translatedSphere = TranslateGS s (0.0, 0.4, 0.0)
ghci> displayRegion translatedSphere
```




```
ghci> c = CubeGS 1
ghci> rotatedCube = RotateGS c (0.0, 0.0, 45)
ghci> displayRegion rotatedCube
```



Listing 10: Examples of regions being transformed.

4.3.3 Render Intersection

To render an intersection of regions we need to make use of the `StencilBuffer` provided by OpenGL. The basic idea of this buffer is that if you draw over a pixel the value in the `StencilBuffer` corresponding to that pixel will change depending on the stencil function and the stencil operation when stencil testing is enabled. We made the `drawIntersection` to draw the intersection of two regions, say region1 and region2. In the beginning of this function the `StencilBuffer` before this function is called is saved as `prevStencilBuffer`. The `ColorMask` from before this function is called is also saved as `prevColorMask`. Before region1 is drawn we disable the `ColorMask`, enable stencil testing and clear the `StencilBuffer` so that nothing actually gets rendered, only the values in the stencil buffer get changed. So when the first region gets drawn, all the values in the `StencilBuffer` corresponding with the pixels getting drawn will be set to 1. All the other values in the `StencilBuffer` will be equal to 0.

Then there are two cases that we need to be aware of. The first case is when `prevColorMask` was enabled when the `drawIntersection` function was called. This means that we have to draw something. So then we will render region2 only where both `prevStencilBuffer` and the current `StencilBuffer` both have a value equal to 1. After this is done we set the current `StencilBuffer` back to `prevStencilBuffer`. The second case is when `prevColorMask` was disabled. This can be the case if the current intersection is inside another intersection. This means we need to set the correct `StencilBuffer`, the intersection of region1 and region2. To do this we first save the `StencilBuffer` after drawing region1 in `stencilBuffer1`. Then we clear the current `StencilBuffer` and do the same as region1, but now for region2. Then we have a `stencilBuffer2` where all the values will be 1 where region2 was drawn, otherwise 0. Then we put 1 in the current `StencilBuffer` if there is a 1 in both `stencilBuffer1` and `stencilBuffer2`, otherwise 0. Then when the funciton is exited in this case the current `StencilBuffer` will be set to the intersection of region1 and region2.

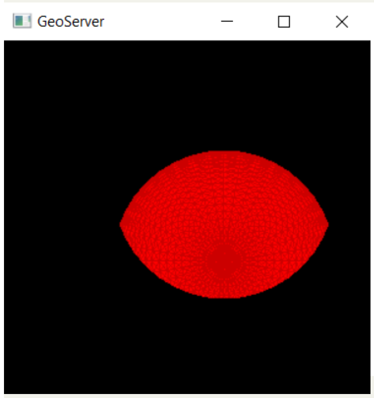
The pseudo code of how the `drawIntersection` function was implemented can be found in Listing **??**. Examples of renders of intersections of regions can be seen in Listing 12.

```
drawIntersection :: Draw() -> Draw() -> Draw()
drawIntersection drawAction1 drawAction2 = do
  -- Get the previous state and save it
  -- this is for the stencilFunc, stencilOp and stencilTest
  prevState <- getPrevState
  prevColorMask <- colorMask
  prevStencilBuffer <- readStencilBuffer

  -- Check if prevColorMask is Disabled
  prevColorMaskIsDisabled = isColorMaskDisabled prevColorMask
  stencilTest $= Enabled
  colorMask $= Disabled
  -- Get first mask
  clear [StencilBuffer]
  stencilFunc $= (Always, 1)
  drawAction1
  stencilBuffer1 <- readStencilBuffer

  -- Now you know something really has to be drawn
  when (not prevColorMaskIsDisabled) $ do
    intersectionStencilBuffer <- computeIntersection prevStencilBuffer stencilBuffer1
    liftIO $ do
      colorMask $= prevColorMask
      StencilBuffer $= intersectionStencilBuffer
      -- Final rendering based on the intersection result
      stencilFunc $= (Equal, 1)
    drawAction2
    -- restore the previous stencil buffer
    StencilBuffer $= prevStencilBuffer

  -- if color mask is disabled we need to get the stencil buffer
```

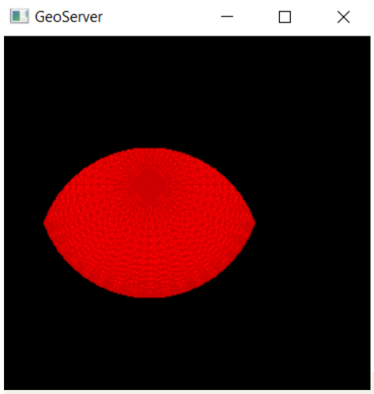
```
  when prevColorMaskIsDisabled $ do
    -- Get second mask
    clear [StencilBuffer]
    stencilFunc $= (Always, 1)
    drawAction2
    stencilBuffer2 readStencilBuffer
    -- Compute the intersection of the two stencil buffers
    intersection1And2StencilBuffer <- liftIO $ computeIntersection (width, height) stencilBuffer1 stencilBuffer2
    -- the intersection1And2StencilBuffer will be set after exiting this function in this case
    writeStencilBuffer intersection1And2StencilBuffer

  -- Restore the previous state
  state $= prevState
  colorMask $= prevColorMask
```

Listing 11: Pseudo code of how the intersection of regions is rendered.

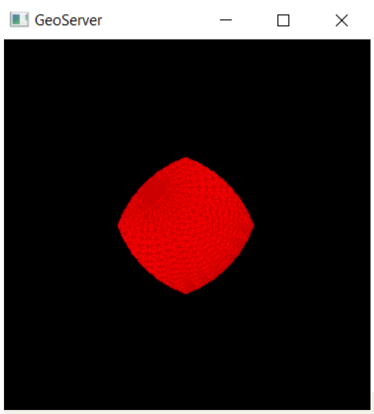
```
ghci> sphere1 = SphereGS 0.6
ghci> translatedSphere2 = TranslateGS sphere1 (0.2, 0.2, 0)
ghci> translatedSphere3 = TranslateGS sphere1 (0.2, -0.2, 0)
ghci> translatedSphere4 = TranslateGS sphere1 (-0.2, -0.2, 0)
ghci> translatedSphere5 = TranslateGS sphere1 (-0.2, 0.2, 0)
ghci> intersection1 = IntersectionGS translatedSphere2 translatedSphere3
ghci> intersection2 = IntersectionGS translatedSphere4 translatedSphere5
ghci> intersectionUlt = IntersectionGS intersection1 intersection2
ghci> displayRegion intersection1
```




```
ghci> displayRegion intersection2
```




```
ghci> displayRegion intersectionUlt
```



Listing 12: Examples of renders of intersections of regions.

4.3.4 Render Outside

To render only the outside of a region we need to use the `StencilBuffer` again like when rendering the intersection of regions. To render the outside of a region we have made the `drawOutside` function. In the beginning of this function the `StencilBuffer` before this function is called is saved as `prevStencilBuffer`. The `ColorMask` from before this function is called is also saved as `prevColorMask`. There are two cases again like in the `drawIntersection` function, as discussed in Section 4.3.3.

The first case is when `prevColorMask` was enabled. This means something has to be rendered. For this we disable the `ColorMask` and clear the current `StencilBuffer`. Then we draw the region we need to draw the outside of. The `StencilBuffer` will be filled with 1's where the region is and on the other pixels there will be 0's. Then we change all the 1's to 0's and all the 0's to 1's in the `StencilBuffer`. Then we fill the `StencilBuffer` with 1's where the current `StencilBuffer` and the `prevStencilBuffer` have 1's. Otherwise the

`StencilBuffer` is filled with 0's. Then we enable the `ColorMask` again and draw over the whole screen, but only where there are 1's in the `StencilBuffer`. This way the outside of the region will have been drawn.

The second case is when `prevColorMask` was disabled. This can be the case when the current `OutsideGS` is inside of another `OutsideGS` or an `IntersectionGS`. If that is the case then the `StencilBuffer` needs to be updated with the pixels which may be rendered. To get this correct `StencilBuffer`, we disable the `ColorMask` and clear the current `StencilBuffer`. Then we draw the region we need to draw the outside of. The `StencilBuffer` will be filled with 1's where the region is and on the other pixels there will be 0's. Then we change all the 1's to 0's and all the 0's to 1's in the `StencilBuffer`. When the function is exited, the correct `StencilBuffer` will be set.

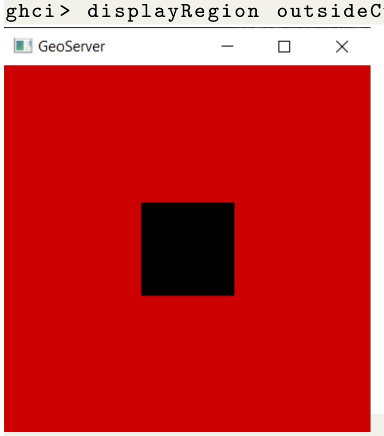
The pseudo code of how the `drawOutside` function was implemented can be found in Listing 13. Examples of renders of the outside of regions can be seen in Listing 14.

```
drawOutside :: Draw() -> Draw()
drawOutside drawAction = do
  -- Get the previous state and save it
  -- this is for the stencilFunc, stencilOp and stencilTest
  prevState <- getPrevState
  prevColorMask <- colorMask
  prevStencilBuffer <- readStencilBuffer

  -- Check if prevColorMask is Disabled
  prevColorMaskIsDisabled = isColorMaskDisabled prevColorMask

  when isColorMaskDisabled $ do
    stencilTest $= Enabled
    colorMask $= Disabled
    clear [StencilBuffer]
    stencilFunc $= (Always, 1)
    drawAction
    -- the gotten stencilBuffer now has to be reversed
    stencilFunc $= (Equal, 1)
    -- parameters are fail, passes, passess
    -- so if it is equal to 1 than it needs to be 0 -> Decr
    -- if it is not equal to 1 make it 1 -> Incr
    stencilOp $= (OpIncr, OpDecr, OpDecr)
    drawOverWholeStencilBuffer

  -- if color mask is not disabled we need to draw
  when (not isColorMaskDisabled) $ do
    stencilTest $= Enabled
    colorMask $= Disabled
    clear [StencilBuffer]
    stencilFunc $= (Always, 1)
    drawAction
    -- the gotten stencilBuffer now has to be reversed
    stencilFunc $= (Equal, 1)
    -- parameters are fail, passes, passess
    -- so if it is equal to 1 than it needs to be 0 -> Decr
```

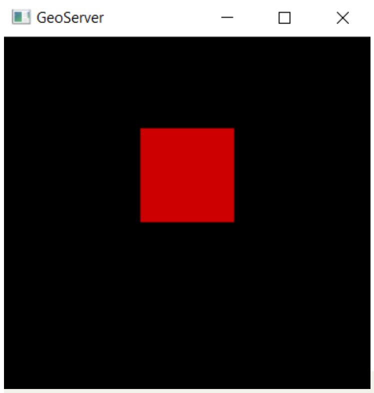
```
    -- if it is not equal to 1 make it 1 -> Incr
    stencilOp $= (OpIncr, OpDecr, OpDecr)
    drawOverWholeStencilBufferr

    -- This part is needed to both handle if the Outside is in root
     or in an IntersectGS
    -- save the current StencilBuffer
    stencilBufferFromOutside <- readStencilBuffer
    -- now intersect it with the previous
    intersectionStencilBuffer <- computeIntersection
    prevStencilBuffer stencilBufferFromOutside
    -- and now set it as the current StencilBuffer
    writeStencilBuffer intersectionStencilBuffer
    -- Now draw everywhere based on this
    colorMask $= Enabled
    stencilFunc $= (Equal, 1)
    drawOverWholeStencilBuffer

    -- restore previous StencilBuffer
    writeStencilBuffer prevStencilBuffer

  -- Restore the previous state
  state $= prevState
  colorMask $= prevColorMask
```

Listing 13: Pseudo code of how the outside of regions is rendered.

```
ghci> cube1 = CubeGS 0.5
ghci> outsideCube = OutsideGS cube1
ghci> translatedOutsideCube = TranslateGS outsideCube (0, 0.25, 0)
ghci> outsideTranslatedOutsideCube = OutsideGS
    translatedOutsideCube
ghci> displayRegion outsideCube
```




```
ghci> displayRegion outsideTranslatedOutsideCube
```



Listing 14: Examples of renders of outside of regions.

4.3.5 Render Union

To render the union of two regions we have made the `drawUnion` function. We need to be aware of the same two cases as in the `drawIntersection` and `drawOutside` functions. If `prevColorMask` was enabled then we simply draw the two regions. However, if `prevColorMask` was disabled we get the two `StencilBuffer` after drawing the first region and then the `StencilBuffer` after drawing the second region. We save them as `stencilBuffer1` and `stencilBuffer2` respectively. We then put 1's in the `StencilBuffer` if `stencilBuffer1` or `stencilBuffer2` have a 1 in that position. Otherwise we put a 0 in the `StencilBuffer`. This way the `StencilBuffer` will be set correctly after rendering the union of the two regions.

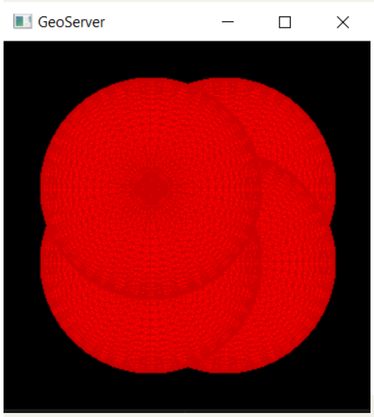
The pseudo code of how the `drawUnion` function was implemented can be found in Listing 15. Examples of renders of union of regions can be seen in Listing 16.

```
drawUnion :: Draw() -> Draw() -> Draw()
drawUnion drawAction1 drawAction2 = do
    prevColorMask <- colorMask
    -- Check if prevColorMask is Disabled
    prevColorMaskIsDisabled = isColorMaskDisabled prevColorMask
    when isColorMaskDisabled $ do
      -- Get the previous state and save it
      -- this is for the stencilFunc, stencilOp and stencilTest
      prevState <- getPrevState
      -- Get first mask
      clear [StencilBuffer]
      stencilFunc $= (Always, 1)
      drawAction1
      stencilBuffer1 <- readStencilBuffer
      -- Get second mask
      clear [StencilBuffer]
      stencilFunc $= (Always, 1)
      drawAction2
      stencilBuffer2 <- readStencilBuffer
```

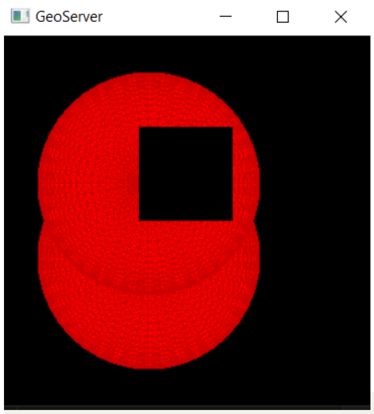
```haskell
      -- Compute the union of the two stencil buffers
      unionStencilBuffer <- computeUnion stencilBuffer1 stencilBuffer2
      -- write the unionStencilBuffer to the StencilBuffer
      writeStencilBuffer unionStencilBuffer
      -- Restore the stencil test state
      colorMask $= prevColorMask
      -- Restore the previous state
      state $= prevState

    -- if color mask is not disabled we need to draw
    unless isColorMaskDisabled $ do
      drawAction1
      drawAction2
```

Listing 15: Pseudo code of how the union of regions is rendered.

```haskell
ghci> sphere1 = SphereGS 0.6
ghci> translatedSphere2 = TranslateGS sphere1 (0.2, 0.2, 0)
ghci> translatedSphere3 = TranslateGS sphere1 (0.2, -0.2, 0)
ghci> translatedSphere4 = TranslateGS sphere1 (-0.2, -0.2, 0)
ghci> translatedSphere5 = TranslateGS sphere1 (-0.2, 0.2, 0)
ghci> union1 = UnionGS translatedSphere2 translatedSphere3
ghci> union2 = UnionGS translatedSphere4 translatedSphere5
ghci> unionUlt = UnionGS union1 union2
ghci> cube1 = CubeGS 0.5
ghci> outsideCube = OutsideGS cube1
ghci> translatedOutsideCube = TranslateGS outsideCube (0, 0.25, 0)
ghci> intersectionOutsideAndUnion = IntersectionGS translatedOutsideCube union2
ghci> displayRegion unionUlt
```




```haskell
ghci> displayRegion intersectionOutsideAndUnion
```



Listing 16: Examples of renders of union of regions.

4.3.6 Adding colors to the regions

OpenGL allows for setting the color of rendering objects by applying the color function on a particular color (in our case Color3 to represent RGB triplets) before rendering the object. Setting the color will be done when rendering primitive regions such as cubes, spheres and cones. However to avoid passing this color as a parameter to all the region constructs when parsing the AST, we chose to use the StateT IO monad. StateT is a monad transformer that offers to combine the functionality of the State monad with another monad (IO in this case). For this we defined a new data type `DrawState` as shown in listing 18. It consists of two colors one for the outline and one for filling in the region. Next we defined a new type alias `Draw a` which will be used to type all the draw functions specified earlier. Functions that have the `Draw()` type have DrawState as their state, perform IO actions and do not produce a meaningful value (as specified by the unit type ()).

```
color $ Color3 (1 0 0)
renderObject Solid (Cube 0.2)
```

Listing 17: Basic example of setting color before rendering an object

```
data DrawState = DrawState {
  fillColor :: Color3 GLdouble,
  outlineColor :: Color3 GLdouble
} deriving (Show)

type Draw a = StateT DrawState IO a
```

Listing 18: The color state monad

Since the draw functions (e.g. `drawCone`) have a return type of `Draw()`, we still need to be able to get the display callback out of it. For this we define a function `runDraw` as shown in listing 19. `runDraw` will evaluate the state on a given draw action (such as `drawCone`) and return a `displayCallback` which is a value of type `IO()`.

```
runDraw :: DrawState -> Draw a -> IO a
runDraw initialState drawAction = evalStateT drawAction
    initialState
```

Listing 19: The runDraw function

Finally we define a function `drawWithOutline` which gets the current draw state colors and sets the colors when rendering the object's outline and interior as shown in listing 20. It renders the object twice. First time with polygon mode set to line so it only draws the outline and with the outline color. Second time with polygon mode set to fill with the fill color. The `drawWithOutline` function gets applied on the `renderObject` function within the `drawCone`, `drawSphere` and `drawCube` functions.

```
drawWithOutline :: DisplayCallback -> Draw()
drawWithOutline renderObject = do
  state <- Control.Monad.State.get
  liftIO $ do
    color $ fillColor state
    renderObject

  liftIO $ do
    -- Set the polygon mode to line to draw the outline
    polygonMode $= (Line, Line)
    color $ outlineColor state
    renderObject
    -- Reset the polygon mode to fill
    polygonMode $= (Fill, Fill)


drawCone :: GLdouble -> GLdouble -> Draw()
drawCone base height = drawWithOutline $ renderObject Solid (Cone
    base height 40 40)
```

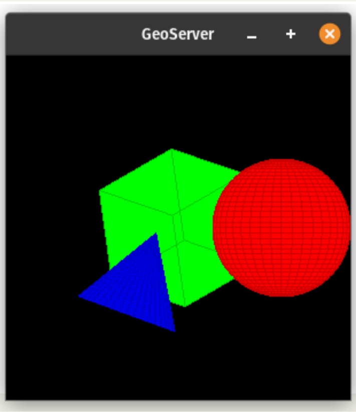
Listing 20: The drawWithOutline function and how it's used in drawCone

To abstract away creating DrawStates and outline colors we have implemented the `drawingWithCustomColors` and `setColor` functions. The code is shown in listing 21. `drawingWithCustomColors` is a variation of `drawing`. Instead of drawing a single region in a default hard coded color, it accepts a list of tuples consisting of a `DrawState` and a `Region`. This means the user can construct a `Region` and combine it in a tuple with a `DrawState`. To simply construct a `DrawState`, the user can call `setColor` on a RGB triplet and an appropriate `DrawState` will be constructed where the fill color corresponds to the RGB triplet and the outline color is set to 80% of the fill color. An example of this usage is shown in listing 22

```haskell
setColor :: GLdouble -> GLdouble -> GLdouble -> DrawState
setColor r g b = DrawState (Color3 r g b) (scaleColor3 0.8 (Color3 r g b))


drawingWithCustomColors :: [(DrawState, Region)] -> DisplayCallback
drawingWithCustomColors regions = do
    clear [ ColorBuffer, DepthBuffer, StencilBuffer]
    -- This is needed for intersections
    stencilTest $= Enabled
    clear [StencilBuffer]
    colorMask $= Color4 Disabled Disabled Disabled Disabled
    stencilFunc $= (Always, 1, 0xFF)
    stencilOp $= (OpReplace, OpReplace, OpReplace)
    drawOverWholeStencilBuffer
    colorMask $= Color4 Enabled Enabled Enabled Enabled
    stencilTest $= Enabled
    -- until here
    mapM_ (\(drawState, region) -> do
      runDraw drawState (foldRegion renderAlgebra region)
      ) regions
    flush
```

Listing 21: Implementation of drawingWithCustomColors and setColor function

```haskell
ghci> cube   = RotateGS (CubeGS 0.6) (20, 45, 10)
ghci> sphere = RotateGS (TranslateGS (SphereGS 0.4) (0.6, 0.2, 0)) (90, 0, 0)
ghci> cone = TranslateGS (RotateGS (ConeGS 0.3 0.5) (-90, 20, 100)) (-0.3, -0.5, 0)
ghci> coloredCube = ((setColor 0 1 0), cube)
ghci> coloredSphere = ((setColor 1 0 0), sphere)
ghci> coloredCone = ((setColor 0 0 1), cone)
ghci> regions = [coloredCube, coloredSphere, coloredCone]
ghci> displayRegionsWithCustomColors regions
```



Listing 22: Example of constructing and displaying colored regions

5 Reflection

It was hard to work with OpenGL. You are able to find examples of OpenGL, but most of these are in C and not in Haskell. This would have not really been an issue if OpenGL functions in C would have a respective function in Haskell, but this was not always the case. Secondly, we had to learn how to render objects. Most of it was straightforward, but learning to work with the stencil buffer, for example, took some time.

GLUT offered a simple way to render some three-dimensional objects. So we used that to render our spheres, cubes and cones. However the `renderObject` function offered by GLUT returns a display callback which exists in the two-dimensional pixel space. Calculating the intersection and outside for three-dimensional objects using only pixels is very hard. Alternatively we could have constructed cubes by modeling it with square faces and then calculate the intersection and outside in the three-dimensional model space before rendering it.

If we had more time, we would have liked to let `IntersectGS` and `OutsideGS` work in three dimensions, not only in two dimensions. The reason we were not able to let it work in three dimensions is because we use the stencil buffer to render regions of the type `IntersectionGS` and `OutsideGS`. Maybe with depth testing this could be resolved. We tried to do this with depth testing, but we did not succeed to correctly implement this.

In the following link you can find the repository we used to make this project: `https://github.com/ArsenBilyalov/CSAI-3D-GeoServer`.

References

[1] JasonDagit ChrisDornan IanLynagh and SvenPanne. *GLUT*. 2024. URL: `https://hackage.haskell.org/package/GLUT`.

[2] Sven Panne and JasonDagit. *OpenGL Haskell*. 2024. URL: `https://hackage.haskell.org/package/OpenGL`.