ComplexPlane.cpp

**ComplexPlane::ComplexPlane(int pixelWidth, int pixelHeight)**

* Assign **m\_pixelWidth** with the parameter values
* Calculate and assign the aspect ratio of the monitor, **m\_aspectRatio**
  + height / width
  + Be careful of integer divide
  + Our object needs to know this so the plane is not distorted
* Assign **m\_plane\_center**with **{0,0}**
* Assign **m\_plane\_size**with **{BASE\_WIDTH, BASE\_HEIGHT \* m\_aspectRatio}**
* Assign **m\_zoomCount** with **0**
* Assign **m\_State** with **State::CALCULATING**to be ready for the initial screen
* Initialize **VertexArray**
  + We will use this to draw a color for each pixel
  + Set its primitive type to **Points**
  + Resize it to**pixelWidth\* pixelHeight**

**void ComplexPlane::draw(RenderTarget& target, RenderStates states) const**

* This function only needs one line of code:
  + **target.draw(m\_vArray);**

**void complexPlane::updateRender()**

* If **m\_State**is **CALCULATING**
  + Create a double for loop to loop through all pixels in the screen height and width
    - Use **j** for **x** and **i** for **y**
      * **Note:**  be careful not to transpose these!
    - Set the **position** variable in the element of **VertexArray** that corresponds to the screen coordinate **j,i**
      * This involves mapping the two-dimensional position at **j,i**to its one-dimensional array index:
        + **vArray[j + i \* pixelWidth].position = { (float)j,(float)i };**
    - Use **ComplexPlane::mapPixelToCoords**to find the **Vector2f** coordinate in the complex plane that corresponds to the screen pixel location at **j,i**
    - Call **ComplexPlane::countIterations**with the **Vector2f** coordinate as its argument and store the number of iterations
    - Declare three local **Uint8** variables **r,g,b**to store the RGB values for the current pixel
      * **Uint8** is an alias for **unsigned char**
    - Pass the number of iterations and the RGB variables into **ComplexPlane::iterationsToRGB**
      * This will assign the RGB values by reference
    - Set the **color**variable in the element of **VertexArray** that corresponds to the screen coordinate **j,i**
      * **vArray[j + i \* pixelWidth].color = { r,g,b };**
  + Set the state to **DISPLAYING**

**void ComplexPlane::zoomIn()**

* Increment **m\_zoomCount**
* Set a local variable for the x size to **BASE\_WIDTH \* (BASE\_ZOOM to the m\_ZoomCount power)**
* Set a local variable for the y size to **BASE\_HEIGHT \* m\_aspectRatio \* (BASE\_ZOOM to the m\_ZoomCount power)**
* Assign **m\_plane\_size**with this new size
* Set **m\_State** to **CALCULATING**

**void ComplexPlane::zoomOut()**

* Same as **zoomIn,** just decrement **m\_zoomCount** instead of incrementing it

**void ComplexPlane::setCenter(Vector2i mousePixel)**

* Use **ComplexPlane::mapPixelToCoords**to find the **Vector2f** coordinate in the complex plane that corresponds to the screen pixel location
* Assign **m\_plane\_center**with this coordinate
* Set **m\_State** to **CALCULATING**

**void ComplexPlane::setMouseLocation(Vector2i mousPixel)**

* Use **ComplexPlane::mapPixelToCoords**to find the **Vector2f** coordinate in the complex plane that corresponds to the screen pixel location
* Assign **m\_mouseLocation** with this coordinate

**void ComplexPlane::loadText(Text& text)**

* Use a **stringstream** and the corresponding member variables to create the following output:
* **Note:**  Cursor should update live as the user moves the mouse.  Center should only update after they click.

**size\_t ComplexPlane::countIterations(Vector2f coord)**

* Count the number of iterations of the set for the given coordinate as specified above

**void ComplexPlane::iterationsToRGB(size\_t count, Uint8& r, Uint8& g, Uint8& b)**

* Map the given iteration **count** to an **r,g,b**color and assign the given reference variables
* You are free to create and experiment with your own color scheme
* You may want to start with gray scale, where **r,g,b** are always the same value in the range [0,255]
* I used the following strategy:
  + At **MAX\_ITER** I colored the pixel black **{0,0,0}**
  + Between [0 : MAX\_ITER) I divided the colors into 5 regions:
    - Purple / blue for low iteration counts
    - Turquoise
    - Green
    - Yellow
    - Red for high iteration counts
  + You can create a color "sliding" effect to differentiate more colors by adding or subtracting the iteration count to one color within a region
* You can experiment with HSL color mapping to see which values to assign for each region
* Set S to 100% and L to 50% and slide the H:
  + [Color MappingLinks to an external site.](https://www.w3schools.com/colors/colors_hsl.asp)

**Vector2f ComplexPlane::mapPixelToCoords(Vector2i mousePixel)**

* The purpose of this function is to map a pixel location on your monitor to a coordinate in the complex plane
* The default range of the complex plane is:
  + real => [-2, 2]
  + imaginary => [-2, 2]
  + The range for imaginary will change depending on the aspect ratio of the monitor
  + These values are stored in **m\_plane\_size**
* Assuming your monitor is something like 1080p, the default range of the display pixels are:
  + x => [0, 1920]
  + y => [1080, 0]
* Thus, if the user clicks near the middle of the screen, for example at pixel location (960, 540), this should map to (0,0) on the complex plane
* The general formula to map a value **n**from range **[a,b]** into range **[c,d]** is
  + **((n  - a) / (b - a)) \* (d - c) + c**
* In the example above,
  + x = 960 would map to:  ((960 - 0) / (1920 - 0)) \* (2 - - 2) + (-2) == 0
  + y = 540 would map to:  ((540 - 1080) / (0 - 1080)) \* (2 - - 2) + (-2) == 0
* The range boundaries of the pixel coordinates will not change
* Changing zoom levels or changing the center will change the range boundaries of the complex plane
  + The magnitude (d - c) is always equal to either **m\_plane\_size.x** or **m\_plane\_size.y**, depending on which direction you are calculating
  + The offset of +c is always equal to either **(m\_plane\_center.x - m\_plane\_size.x / 2.0)**or **(m\_plane\_center.y - m\_plane\_size.y / 2.0)**
* Below is a visual of the mapping algorithm: