Project Abacus

Progress Report

Our group has completed all of the phase 0 tasks, including specification, CRC cards, scenario, and a basic skeleton program.
The code is able to generate detailed imagery for pre-configured “graphs”, in both 2D and 3D with examples provided.
There is also a rigid plan for an Abstract Syntax Tree, which will be the method used to store expressions in order to graph them.

Specification

A User should be able to:

- actually evaluate basic expressions like 3 + 5
- graph functions from $\mathbb{R}$ to $\mathbb{R}$ and restrict its domain
- graph functions in 3D, so $\mathbb{R}^2 \to \mathbb{R}$ and again be able to restrict domains
- graph implicit functions, i.e. functions of the form $f(x, y) = 0$ and $f(x, y, z) = 0$.
- plot multiple graphs on the same set of axes
- do operations on the functions they define, i.e. if they have a function $f(x)$ and $g(x)$, they should be able to plot $f(x) + g(x), f(x) - g(x), f(x)g(x), \frac{f(x)}{g(x)}, f(g(x))$
- zoom in and out of graph
- be able to rotate viewpoint around for 3D graphs

Possible extensions:

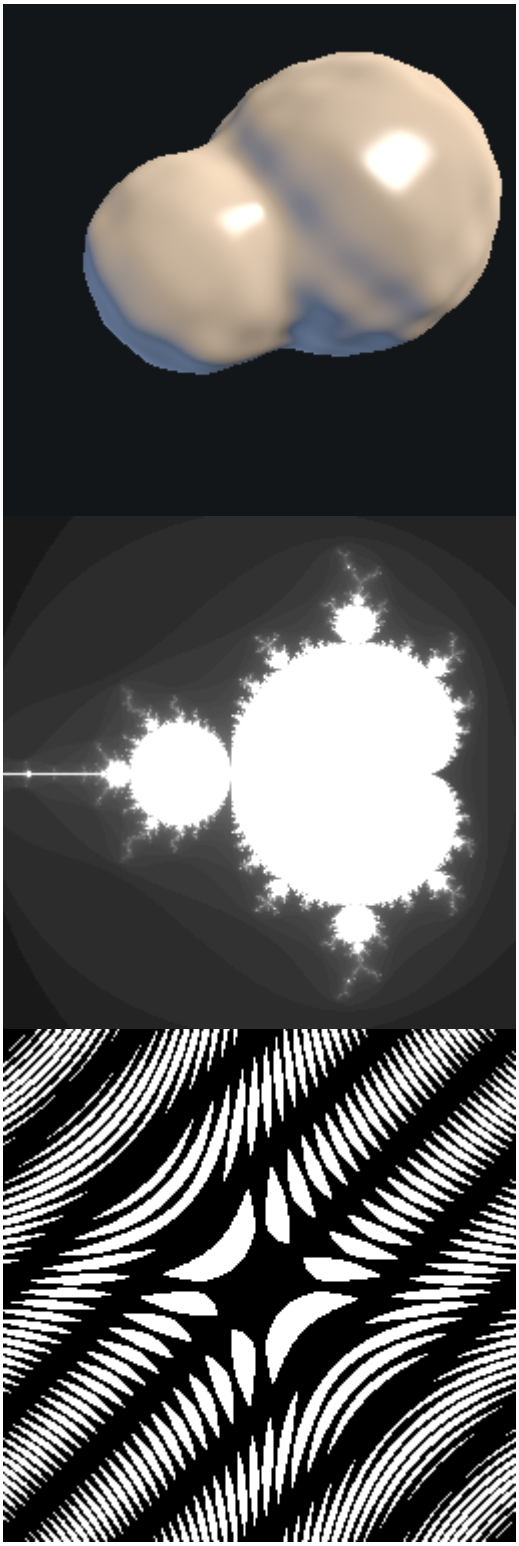
- Plotting parametric curves and surfaces
- Plotting complex functions
- Plotting vector fields
- Finding derivatives and integrals

Scenario Walk-through

A user inputs an expression like `f(x) = x^2 + 5`. First an empty `Backend.Axes` object is created. This is passed on to `Backend.ExpressionReader`. The `Backend.ExpressionReader` converts `x^2 + 5` into a list `[“x”, “^”, “2”, “+”, “5”]` and passes that on to `Backend.ExpressionCreator`. `Backend.ExpressionCreator` will convert all the variables, operators and numbers into appropriate `Expressions` (`Backend.NumberExpression`, `Backend.VariableExpression`, `Backend.OperatorExpression`) and combines them into an Abstract Syntax Tree, then create a `Backend.FunctionExpression` that stores the function name “f” along with the expression that it evaluates. `Backend.ExpressionReader` will then return this `Backend.FunctionExpression` which will then be added to the `Backend.Axes` object.

The `Renderer` takes an `Backend.Axes` (currently just an `Backend.Expression`), a `Viewpoint`, and other parameters for the desired image (such as size, scales of axes, etc), then generates an int[] array representing pixels of an image. `RendererUseCase` will pass the parameters of the image to `Renderer` which will then convert the pixel array into an image and save it to a file.

Renderer Demo Results



What each group member has been working on and plans to work on next?

AST/Backend.Expression implementation: Dihan, Rishibh, Ted, Omar

- Creating the Backend.Expression class that represents a given function as an abstract syntax tree.
- For a given input string representing an expression, creating the Backend.ExpressionReader and Backend.ExpressionCreator classes to build the function AST.
- To-do

    - AST operations like function composition
    - Other mathematical operations e.g. trigonometry, logarithms
    - Clean up code
    - Handling brackets more appropriately
    - Handling constants like pi and e
- Graphics/Graphing: Kevin, Louis, Binseong, Andrew
    - Created basic 2D implicit Grapher
    - 3D SDF Raymarcher
    - Will try extending Raymarcher to arbitrary implicit functions
    - 3D - adjustable lighting, shadows etc
    - 2D - color gradients etc
    - Will try OpenGL or GUI
- ExpressionReader: Omar and Ted
- Testing: Omar, Binseong

What has worked well so far with your design as you have started implementing the code?

- Having a recursive structure for Expressions works really well
- Having modularity between the ASTs and Renderer classes means that we can have groups working on them independently

Have you clearly indicated at least one open question your group is struggling with, so that your TA can address it in their feedback for phase 0?

- Categorizing classes by clean architecture (especially with controllers and use cases)