Type Assistance for Graphics System (T.A.G.S.)

Vision:

In graphics programming there are often times when very subtle and nearly undetectable bugs appear, often when computing between different coordinate spaces or reference points. The goal of this project was to create a type system and programming language to eliminate incorrect transformations between coordinate spaces and reference spaces.

The method I used to address this is the titular "Tags" system. Each tag consists of a "space" and a "value". For example a space could be "coordinate system", "reference point" etc. The value then declares the corresponding value, such as "spherical" for coordinate system, or "world" for reference point. Operations between two tagged expressions will only be valid if any matching space has a matching value.

Summary of progress:

I reformatted the tag types to be more complex and better support pattern matching.

The basic structure changed to "Tag of string * string" and "Tagged expr of expr * tag". This was reflected in the type system as well.

Maybe the most difficult task was to implement the tag type in the lexer and parser. There were some issues with regular expressions on strings. After getting some great help in OH I got the format below to work.

```
e1 = expr; TAGGED; LBRACK; space = STRING; EQUALS; value = STRING; RBRACK
```

Now basic tags were working of the form:

```
TaggedExpr (Int 1, Tag ("space", "cart"))
```

I had originally intended to implement a list of Tags attached to each TaggedExpression, and implemented this in the AST and main function, however I was unable to implement this in the lexer, so I eventually abandoned this idea and simply used one tag per expression. (This would be a great future direction for the project.)

The next big step was to typecheck less naively. The new tag structure made this more complex. I set up a helper function to check if two tags are compatible: if they share the same space, they must share the same value. It is worth noting that this implementation is "inclusive", as a function is valid as long as the expressions do not have conflicting tags. An “exclusive” system would make all function invalid unless the expressions had matching tags.

I then implemented a “typeof” function which threw an error on type mismatch and returned the type otherwise. I also implemented binary operation typing on TaggedExpressions.

Activity breakdown:

I did all the work for this project.

Productivity analysis:

I am very satisfied with the amount of work I got done in this final sprint. I feel that it was much more productive than any prior sprint, and I was able to get the system to a point which meets my initial goals.

My estimates on how long the goals from last sprint would take were quite off, implementing the new type system and typechecking logic took quite a while, and was the bulk of this sprint. This also involved quite a few redesigns of my system implementation.

Grade:

I accomplished a mix of the elements of my goals from last sprint, both from the “Good” and “Excellent”. I would rate myself as “Excellent” for this sprint, since I spent a lot of time and made progress that I am very satisfied with. I think that I met the original vision I set out for this project.

Future directions:

If instead of single tags on each expression there were a list of tags, many options would open up. The type check system could become much more useful, and there could be interesting operations involving nested tags.

While this originally had just the goal of addressing geometry bugs in graphics programming, I’ve realized that this can be applied much more broadly. Since all space and values are just strings defined by the user this is extensible to many other fields. Some interesting ones to explore would be unit conversion, and maybe even philosophical logical reasoning.