Final Project

I had spent a lot of time reading Math stack exchange, and I had been repeatedly frustrated by phrases like “it is easy to see that”, when most of the time it was not in fact easy to see.
In the past, when I took DXARTS 450, I also focused my final project on comedy. So, I thought why not try something similar again.
A vision came to me: I should have the audience see what it’s like to read some of these posts.

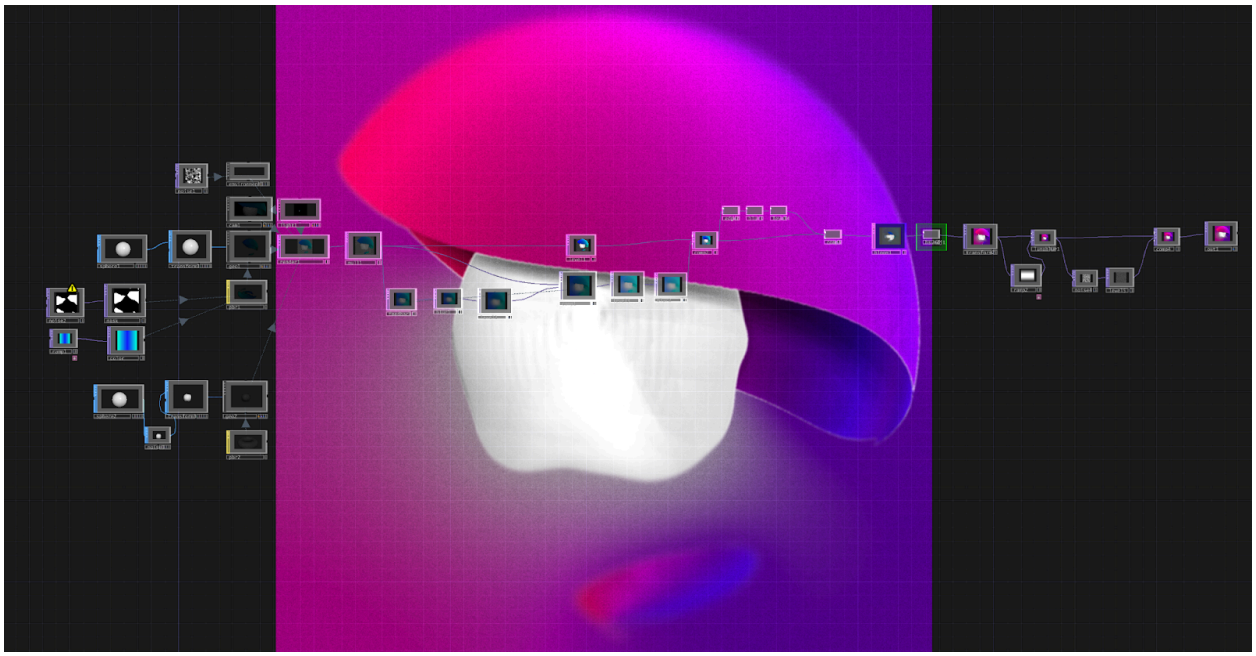
So, two things were clear, I needed to get input from the audience, and somehow, I needed to return an output that sounded similar, authentically like the MSE posts I had read that contain the words “obvious” or “trivial” or “it is easy to see that”.

The eventual answer was to fine-tune GPT2 on MSE posts scraped using the stack exchange API. One such function from that is here:

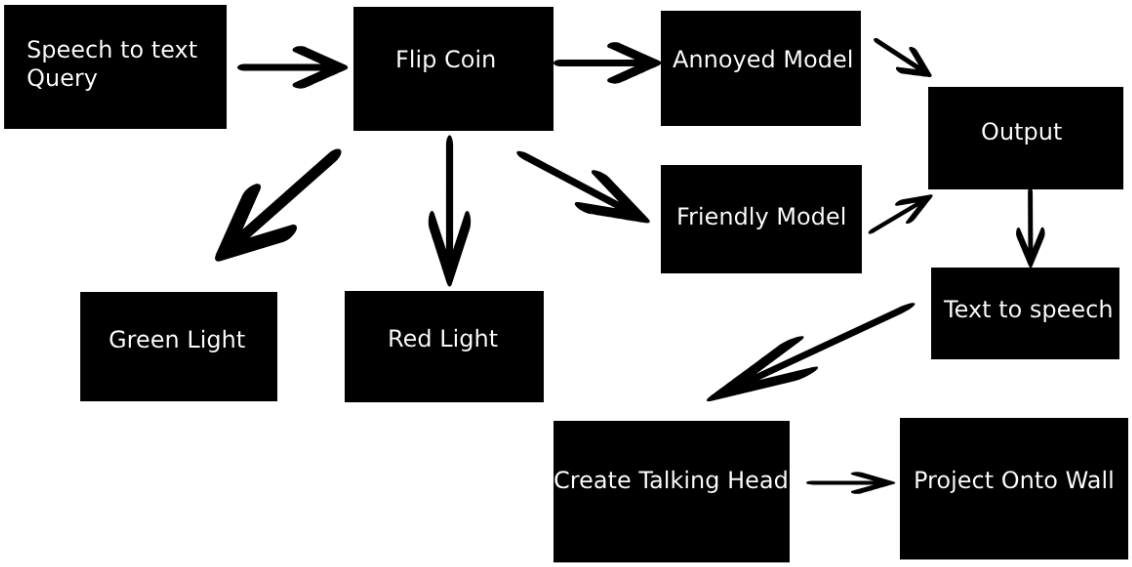
```
def get_obvious_answers_math(max_pages=5, page_size=100):
  url = "https://api.stackexchange.com/2.3/search/advanced"
  all_bodies = []

  for page in range(1, max_pages + 1):
     params = {
        "order": "desc",
        "sort": "relevance",
        "q": "obvious easy trivial",  # Updated to include multiple keywords
        "site": "math.stackexchange",
        "is_answer": True,
        "filter": "withbody",
        "page": page,
        "pagesize": page_size
     }

     response = requests.get(url, params=params)
     response.raise_for_status()  # Raises an error if the request failed
     data = response.json()

     # Extract and clean the 'body' field from each item
     bodies = [clean_stack_exchange_post(item["body"]) for item in data.get("items", [])]

     # Filter the cleaned bodies based on your custom filter
     filtered_bodies = [body for body in bodies if body_is_rude(body)]

     # Add the filtered bodies to the overall list
     all_bodies.extend(filtered_bodies)

     # Check if there are more pages available
```

```
    if not data.get('has_more', False):
        break  # Stop if there are no more results

  sentences = [replace_punctuation_with_space(sentence) for body in all_bodies for sentence
in sent_tokenize(body)]
  return sentences
```

After I had the original data, I had to get to work with TouchDesigner. From the midterm, Professor Luna Castillo gave me some great advice that some might find my original idea, which was to have a human head speak the output, offensive. I understood this point and actually it was something I myself had thought before I presented it. So, I decided to use a cool looking sphere with a cool background floating around it instead. I talked about this in a past homework assignment, but this is how it ended up looking before I implemented it with everything else:



With what was speaking figured out, I needed to figure out the rest of the pipeline. This was the same as I had planned in the midterm, so for reference:

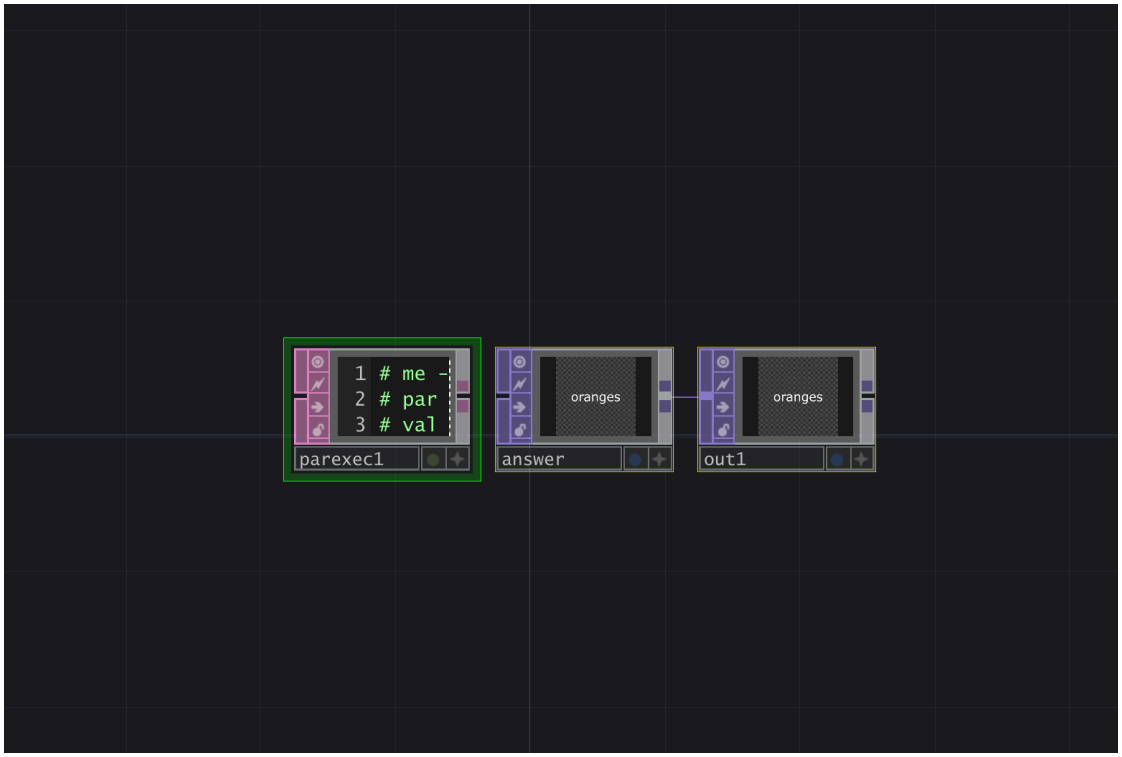


The friendly model was provided as a .toe file in one of the announcements. I used that one-for-one. The annoyed model was much more complicated. I ended up use the hugging face API to fine-tune GPT2, on the MSE answers that I scraped from above. I learned recently that you don't need labeled data to train language models, instead the "labels" is just the next word in the sentence. So I didn't need to label the data or anything like that, and just passed it in, and fine-tuned it with a hugging face API trainer.

```
trainer = Trainer(
  model=model,
  args=training_args,
  train_dataset=tokenized_datasets["train"],
  eval_dataset=tokenized_datasets["test"],
  tokenizer=tokenizer,  # Pass the tokenizer for logging convenience
)
```

Figuring out how to make the tokenized_datasets did take a while though. So that was done.

The next thing I worked on was the text to speech. I tried using the elephant labs, but it just never ended up working for me. So I used openAI whisper instead, which was provided in a .toe file. This was quite an instructive toe file: it taught me how to use the paramexec DAT, which I then used to make my own comps for the friendly model and the angry model. This was the end result for the angry model:



The speech-to-text comp and the others looked very similar. To make the talking sphere audio-reactive, I downloaded the blackhole tool so it would capture my system audio, passed this into an analyze, then got the maximum height of the signal, and used that as the radius of the outer sphere (and took a maximum of that and 0.5 to make it look nicer when nothing is happening).

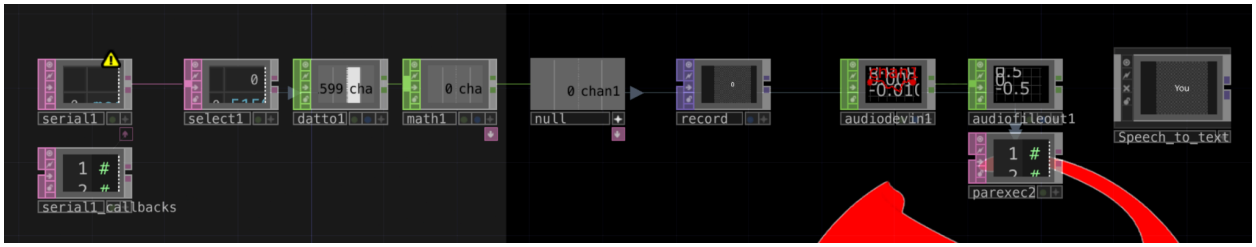
Changing the color was simple, I had a hsvadjust top near the end of the talking ball comp I had made, so I just made a parameter of which color it should be (0 or 1) and then had a simple if statement which would change the color in the hsvadjust.

Visualized here:

So now I just had to deal with getting input from the user. As I said above, I made a .toe file that was a parexec with an API call to openAI whisper that would output what the audio file said.

But, I needed to record input from the user, and more importantly, it had to be seamless: the user could not see what was going on. So I decided to use the arduino we used in class, made the simple touch sensor, and got the number in touchdesigner.

With the number, I could find a threshold so that if the value was greater than this threshold, the user was touching it, and otherwise the user wasn't. Visualized here:



With that settled, I had the recording process finished: with an audiodevicein CHOP set up to record when the user was touching the arduino, I could ask the user to touch and hold the brass sheet, ask a question, and then let go when they were done. Here is a video of that happening: https://drive.google.com/file/d/1sIWgK1PgdvQMpUV-PLUoosEjwK1N7RVY/view?usp=drive_link

Finally, the last part was getting the random number to work. I ended up just using the current time, which was random enough. When the speech to text text field was updated, it would use the current time (mod something) to query one of the models. Then, when the model was done, it would update a text comp called output that is outside of both, and I paired a parexec dat with this output to query the text-to-speech pipeline when the output was updated. This is visualized here:

So with all of that done, I could hide everything, and project the talking ball onto the wall. The user could use the entire pipeline, and again that is shown in the above video.

Everything ended up working as expected, and I really like the way it turned out! I hope you did too!