Project Overview

The primary goal of this project was to learn how to utilize a pre-built AI model from Hugging Face and integrate it into my own Python environment. I also wanted to experiment with a model type I hadn't previously worked with. I chose to work with a text-to-3D model that takes text prompts as input and generates corresponding 3D imagery. This was accomplished using the OpenAI SHAP-E model and documentation from the https://github.com/openai/shap-e repository. The model is based on research and code from Cornell University by Heewoo Jun and Alex Nichol (2023): https://arxiv.org/abs/2305.02463. My objective was to evaluate how accurately the generated 3D objects matched my text prompts and to experiment with model parameters to optimize results for my desired outcomes.

Applications and Impact

Text-to-3D generation has numerous practical applications and potential impacts across various industries. Traditional 3D object creation is both difficult and time-consuming, requiring specialized training in software such as Blender. Since 3D objects are commonly used in visual media like video games, the ability to create 3D assets without traditional modeling expertise could allow creators to allocate more time to other aspects of game development. In practice the models I was able to generate would probably not suffice as finished assets but would work great as temporary assets during development or to quickly prototype 3-D models.

Beyond gaming, text-to-3D technology could benefit film and animation studios, engineering design, educational content creation, and many other fields where 3D visualization is valuable.

Implementation Workflow

The workflow for this application was relatively straightforward, requiring several key steps: downloading and importing necessary packages, setting up variables, defining model parameters, and generating output.

The most challenging aspect of the workflow was the GPU requirement for reasonable performance. Creating 3D objects on a GPU took approximately 1-2 minutes, while CPU processing would require over an hour. Fortunately, I had access to Google Colab's T4 GPU, which enabled quick object generation. Its worth noting however, that I was only able to generate a few 3-D objects before I hit the collabs limit on GPU usage.

```
# Install Shap-e from github
!pip install transformers accelerate -q
!pip install git+https://github.com/openai/shap-e
```

```
pip install diffusers
```

```
[4] # import diffuser pipeline
    from diffusers import ShapEPipeline
    from diffusers.utils import export_to_gif
```

```
[5] # import pytorch for ML tasks
    import torch
    # import neccessary packages from shap-e
    from shap_e.diffusion.sample import sample_latents
    from shap_e.diffusion.gaussian_diffusion import diffusion_from_config
    from shap_e.models.download import load_model, load_config
    from shap_e.util.notebooks import create_pan_cameras, decode_latent_images, gif_widget
```

```
[6] # selecting pre trained model and giving it object name
    ckpt_id = "openai/shap-e"
    # defining hardware for image generation. wants to use GPU but if none available uses CPU
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
[7] # 3D renderer
    xm = load_model('transmitter', device=device)

    # text to 3D AI component
    model = load_model('text300M', device=device)

    # Diffusion process
    diffusion = diffusion_from_config(load_config('diffusion'))
```

Model Set up

The screenshot above shows the necessary steps to prepare the SHAP-E pipeline and model. I've added comments throughout my code to explain the function of each step.

```
batch_size = 4 # how many versions of the 3-D object to create
guidance_scale = 16.0 # higher number -> less creativity, lower number -> gives AI more freedom
prompt = "Knight Sword" # what 3-D object you would like to be rendered

latents = sample_latents(
    batch_size=batch_size,
    model=model,
    diffusion=diffusion,
    guidance_scale=guidance_scale,
    model_kwargs=dict(texts=[prompt] * batch_size),
    progress=True,
    clip_denoised=True,
    use_fp16=True, # change to false for higher quality, slower generation
    use_karras=True, # True is higher quality noise creation
    karras_steps=32, # higher -> more diffusion steps and creates higher quality but slower. bit increments (2, 4, 8, 16, 32, 64)
    sigma_min=1e-3,
    sigma_max=160,
    s_churn=0,
)
```

```
100% 32/32 [01:02<00:00, 1.68s/it]
```

Parameter Configuration and Prompt Input

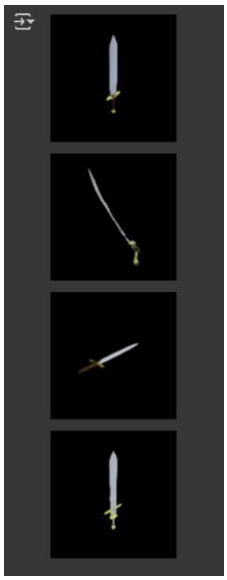
After downloading the required packages, the next code block was used to set model parameters and enter the text prompt. In the screenshot above, I used an example where I generated a "knight sword." The batch size input of 4 means the model will create four distinct variations. The guidance scale, set to 16.0, represents the default value for AI creativity. Several arguments define the model and diffuser type, which were configured in the previous code block. The parameters use_fp16, use_karras, and karras_steps can all be adjusted to modify the quality of the generated 3D objects. In this example, I maintained the default values. Once this code completes execution, the 3D objects are successfully created.

```
render_mode = 'nerf' # you can change this to 'stf' stf is faster but lower quality
size = 128 # this is the size of the renders; higher values take longer to render.  bit increments (2, 4, 8, 16, 32, 64)

cameras = create_pan_cameras(size, device)
for i, latent in enumerate(latents):
    images = decode_latent_images(xm, latent, cameras, rendering_mode=render_mode)
    display(gif_widget(images))
```

Viewing Generated Objects

Next, we run the following code block to view the 3D objects within our Python environment. This code creates small GIF widgets for each batch and displays them as interactive elements. Below is a screenshot of the output. In the notebook, each object appears as a live image allowing viewers to see multiple angles.



Exporting 3D Files

The following section demonstrates how to save these objects as .ply files for more detailed viewing. Once the 3D objects are created and we execute the subsequent code cells, we can export the 3D objects as either .ply or .obj mesh files.

```
# Example of saving the latents as meshes.
from shap_e.util.notebooks import decode_latent_mesh

for i, latent in enumerate(latents):
    t = decode_latent_mesh(xm, latent).tri_mesh()
    with open(f'example_mesh_{i}.ply', 'wb') as f:
        t.write_ply(f)
    with open(f'example_mesh_{i}.obj', 'w') as f:
        t.write_obj(f)
```

```
[ ] from google.colab import files
    import os

    # Download the files to your local machine
    for filename in os.listdir('.'):
        if filename.endswith('.ply'):
            files.download(filename)
```

External Viewing

Using 3dviewer.net, I was able to upload these mesh files and examine the 3D objects in greater detail while experimenting with different orientations. Below is a screenshot of one of the generated "knight swords."

I've included several mesh files with this report for your examination. To view them in detail, visit 3dviewer.net and drag and drop the files onto the webpage.

Final Thoughts

This project successfully demonstrated the implementation of a Hugging Face text-to-3D model in a Python environment. The SHAP-E model proved capable of generating recognizable 3D objects from text prompts, though the quality and accuracy varied depending on the complexity of the requested object and the parameter settings used. Simple requests had good results but more complex requests yielded interesting results that weren't quite usable, like this output of a turtle I tried to get: