Tex-match: command line utility for matching texture maps

Project summary

1 Problem specification and motivation

1.1 About texture maps

Texture maps in 3D graphics are used to define color and high frequency details on 3D models. Although most equate the word "texture" with "diffuse map", which defines the diffuse color of the object, most if not all 3D models use multiple maps. The number and nature of these maps depend on the intended purpose of the model, and in the case of game assets, the engine it was designed for. Due to this diversity, texture maps can be difficult for developers to work with after they have already been generated. In ideal cases, this difficulty stems from making engine-specific formats work with the developer tools, but additional problems may arise due to missing texture files. 3D environments in particular can have texture maps numbering in the hundreds, making it extremely difficult for a human to correct such errors.

1.2 How errors happen

The two most common causes of textures going missing are misnamed files and accidental vertical flipping. The former is self-explanatory: texture files that belong together use a common naming scheme, and if any are misnamed, the only way to find the missing files is to visually match them to their counterparts. On the other hand, vertical flipping generally occurs when textures go through format conversions (such as DDS to PNG). Some formats used inside game engines, like the aforementioned DDS, have to be vertically flipped to be interpreted correctly by the engine. Normally, the flipping-unflipping is handled automatically by the used conversion software, but manual conversions can sometimes result in textures with the wrong orientation.

1.3 The aim of the project

The aim of this project was to automate searching for missing texture maps. In most cases, a human is able to match related texture maps based on visual features, most notably common shapes found in each map at the same location. However, as mentioned before, 3D environments often use hundreds of texture maps. One 3D environment I have previously worked with has over 700 maps, of which several were misnamed, culminating in a long and tedious manual search.

2 Implementation

Tex-match is a command line utility implemented in Python 3 that takes one required argument (the path to the input image) and four optional arguments (two mode flags, path to the directory

to be parsed, path to the output directory). It supports .png, .jpg, .tif and .bmp formats. The bulk of the image processing code uses the OpenCV library and NumPy.

2.1 Loading the images

Using OpenCV's imread method, the images are represented as NumPy arrays. All images are loaded with the `IMREAD_UNCHANGED` flag to preserve the alpha channel for later handling. The command line arguments are parsed using the `argparse` module, and the `glob` module allows the retrieval of supported format images from the specified directory.

2.2 Preprocessing

To understand how the images need to be processed, some examples of texture maps are worth observing.

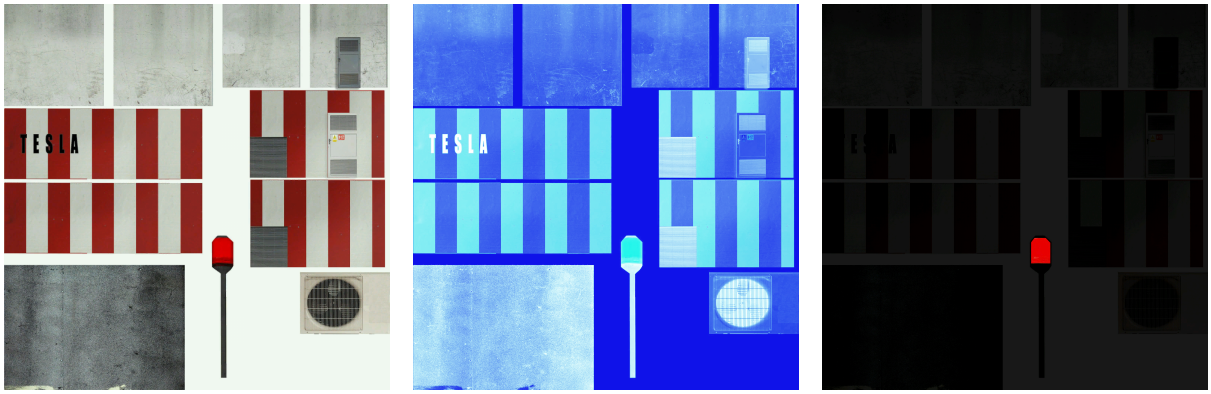
(a) albedo (b) normal+metallic+roughness (c) emissive

Figure 1: Texture maps of an XPlane 11 asset. An example of a shared feature is outlined in red

Figure 1 shows the texture maps of an XPlane object. These are the following:

- albedo map: defines the base color of the object (without shadows and highlights)
- normal+metallic+roughness map: fakes high frequency detail, defines metalness and defines reflectivity, respectively
- emissive map: defines which parts of the model emit light (the whiter the area, the more light is emitted)

It can be seen that the combined normal+metallic+roughness map makes the features shared with the albedo map very apparent to the human eye. However, looking at the emissive map, the similarities are more difficult to pick out. Most objects do not emit light from many surfaces, which makes emissive maps less descriptive than the others. This map in particular is also shared between multiple objects, hence containing details in areas where the other maps are simply padded. A human observer may therefore gloss over the shared features due to the image being visually very different from its counterparts.



(a) diffuse (b) normal (c) emissive

Figure 2: Texture maps of a Prepar3D asset

Figure 2 shows the more basic map setup of a Prepar3D asset:

- diffuse map: defines the color of the object under white light (with shadows and highlights)
- normal map: fakes high frequency detail
- emissive map (RGB): defines the night texture of the object

These maps have very apparent shared features. The parts of the object are nicely separated, only differing in color and luminosity between the three images.

Although these examples are easy for a human observer to interpret, the variance in color and luminosity makes it difficult to find a straightforward matching algorithm. To make the images more uniform, the following steps are taken:

1. Handle the alpha channel, if present. An alpha channel defines transparency in the image and makes its array representation incompatible with most image processing functions, as it exists as a 4th channel alongside the R, G and B channels. For the image to perceptually stay the same, we have to blend it with a solid white background using the equation

$$new = alpha \cdot original + NOT\ alpha \cdot background,$$

   where *alpha* is normalized to [0, 1] range and *background* is 255 for each pixel [1]. The second part can be simplified by scaling the background values instead of the alpha values, leaving $NOT\ alpha \cdot 1$. It is worth noting that the alpha channel can be straight or premultiplied, which affects the equation to be used. However, as there is no way to determine which representation an image uses based on its pixel values, I made the assumption that most images would use the straight representation.

2. Convert the images to grayscale. All further operations require two-dimensional input and there is no additional information to be extracted from the color values.

3. Perform histogram equalization. This ensures that all images are represented in roughly the same way and with sufficient contrast.

4. Upscale lower resolution images to the size of the largest image to ensure uniform scale. Quality is not a concern here as Gaussian blurring will later be applied to all images regardless of the chosen matching method.

In summary, after preprocessing all images are represented as two-dimensional, uniformly scaled NumPy arrays.

2.3 Matching

I implemented two matching methods: feature matching and template matching. The mode can be selected with an optional command line flag when running the program. The default matching mode is template matching due to its speed and accuracy.

Originally, I intended to use feature extraction to detect interesting points in the images, which could later be used to find shared elements. However, the available algorithms are not invariant to luminosity, so the descriptors cannot be used to find matches between the images. The only option is to match the keypoint locations themselves, using the knowledge that shared features must fall to the same location. The SIFT algorithm appeared to be the most appropriate for these types of images, correctly identifying prominent corners and blobs, with the obvious drawback of it being very computationally expensive. Feature matching is achieved through the following steps:

1. Extract the 200 strongest SIFT keypoints from the parsed images and the template.
2. From each keypoint, keep only the location (xy coordinates) and the magnitude. Also create a copy of the keypoints and flip them vertically.
3. For each keypoint in each parsed image, calculate the Euclidean distance to the template keypoints and calculate their magnitude difference.
4. Based on set thresholds, give each image a similarity score that consists of the number of shared features found, the mean Euclidean distance, and the mean magnitude difference. Sort the list of scores in this same hierarchical order so the property with the highest priority is the number of shared features.
5. Select at most 5 best matches (the first 5 elements in the sorted score list).

The approach of template matching turned out to be approximately 10 times faster than feature matching and highly accurate. Template matching is achieved through the following steps:

1. Add a Gaussian blur filter to each image.
2. Perform Canny edge detection with Otsu threshold and dilate the edges with a cross kernel.
3. Due to how OpenCV's template matching function works, the template cannot be the same size as the images we are comparing it to. The input image is therefore trimmed to a smaller size.
4. Overlap the template with each parsed image, which outputs the resulting global maxima. Flip the template along the 0 axis and call the template matching function for each image again to handle vertical flipping.
5. Discard matches whose maxima are below a set threshold.
6. Select at most 5 best matches (the images with the highest maxima).

2.4 Writing results

In both matching modes, the results are written to a plain text file in the specified output path. The file contains the path to the input image, the matching mode the program was run in, and the paths to the found matches. If the list of matches is empty, the result is written to the console without a file being generated.

3 Dependencies and command line usage

To install dependencies, run `$ pip install -r requirements.txt`.

Command line usage:

```
python -m tex_match [-h] [-p PARSE_DIR] [-o OUTPUT_DIR] [-t] [-f] input
```

positional arguments:

input	path to the input texture map

optional arguments:

`-h, --help`	show this help message and exit
`-p PARSE_DIR, --parse-dir PARSE_DIR`	directory to look for matches in
`-o OUTPUT_DIR, --output-dir OUTPUT_DIR`	directory to write matches to
`-t`	use template matching (fast, default)
`-f`	match based on local features (slow)

3.1 Examples

Running the search on a .png image located in /tests/maps, in the default template matching mode:

```
python -m tex_match "input.png"
```

Running feature matching mode:

```
python -m tex_match "input.png" -f
```

If the texture maps are located in a different directory:

```
python -m tex_match "path/to/input.png" -p "path/to/texture maps/" -o
"path/to/output directory/"
```

If the specified output directory does not exist, it will be automatically created.

4 Results and development possibilities

Manual testing showed very high accuracy with both matching methods. Unfortunately, due to the nature of the problem, an automatic testing suite would require creating a database of a large amount of textures, which is outside the scope of this project.

Feature matching is very computationally expensive, taking approximately 35 seconds to process 59 images. Because template matching is highly accurate (often more so than feature matching) and computationally efficient, it was set as the default mode for the program to run in.

In the future, a graphical user interface could be implemented to improve user interaction.

References

[1] "Alpha compositing," Wikipedia. [Online]. Available: (`https://en.wikipedia.org/wiki/Alpha_compositing`)