Procedural and Neural Net Flower Generation

Introduction

This project involves generating a dataset of 3D flower shapes and then training an neural network to generate them. I chose flowers because they are 1) instantly recognizable shapes, 2) relatively easy to generate, and 3) come in a set of distinctive varieties.

To generate the flowers, I wrote a procedure in the Unity game engine utilizing its built-in mesh generation system. To train a neural network to replicate the generation, I use an implicit field decoder paired with a generative adversarial network. In the proceeding sections I elaborate on my process of implementing these components.

Part 1: Generating Flowers

The primary inspiration for my flower generation procedure comes from The Algorithmic Beauty of Plants [PL96], which presents a simple algorithm to arrange petals around a central axis to create a flower:

$$\phi = n * 137.5^\circ, r = c\sqrt{n}$$

Here, ϕ and r are the degree and radius to place the petal, n is the index of the petal, starting from the center-most petal and spiraling outwards, and c is a small scaling constant within the ballpark of 0.01.

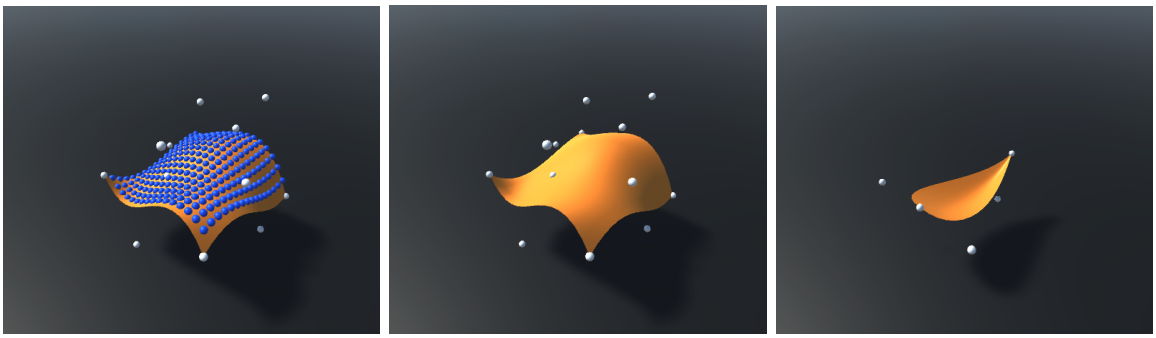
Of course, this algorithm is useless without a means of obtaining petals. Having not taken CS1230 at the time (I'm taking it next fall), I turned to the Unity game engine for mesh generation. Still, having no experience with computer graphics at this point, the official documentation was quite daunting; luckily, I dug up a hidden gem of an online tutorial [Sur03] that explained everything I need to know. Meshes are generated by 1) assigning vertices to coordinates in 3D space and 2) specifying how those vertices are connected by defining a list of indices, where triplets of indices define the vertices to be used in the triangle.

Still, having to generate petals by manually specifying each vertex would be incredibly tedious. An elegant alternative is the bicubic patch (aka bicubic Bezier surface), which resemble a 3D Photoshop "pen" tool in that a small number of control points specify the entire surface. Specifically, a 4x4 grid of control points is used to map points from the unit square onto the surface:

$$p(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) k_{ij}$$

Here, u and v are values from 0 to 1 from the unit square, and $p(u, v)$ is the mapped point on the surface. i and j are indices into the set of control points k, and $B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$. As this is a bicubic patch, n and m are set to 3.

Generating a petal mesh with the bicubic patch is simply a matter of mapping an arbitrarily large 2D grid of values ranging from 0 to 1 (I chose 16 x 16) into a set of 16 x 16 vertices, and then drawing triangles between those vertices as if the vertices were arranged in a flat plane.

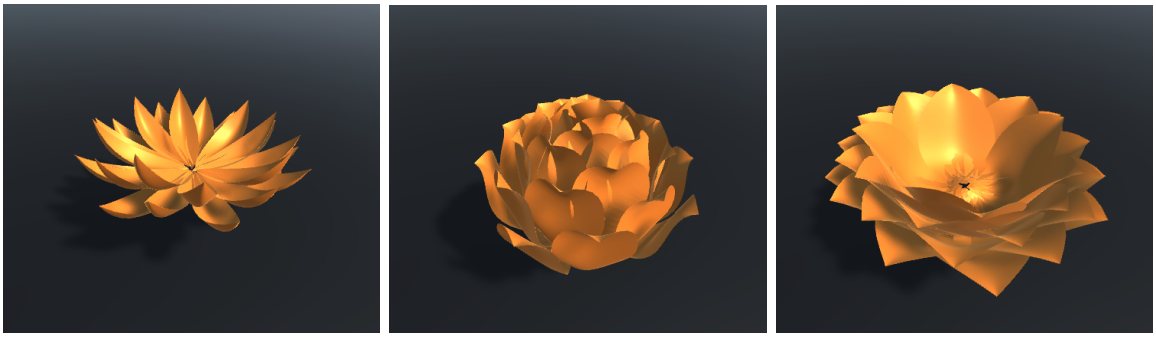


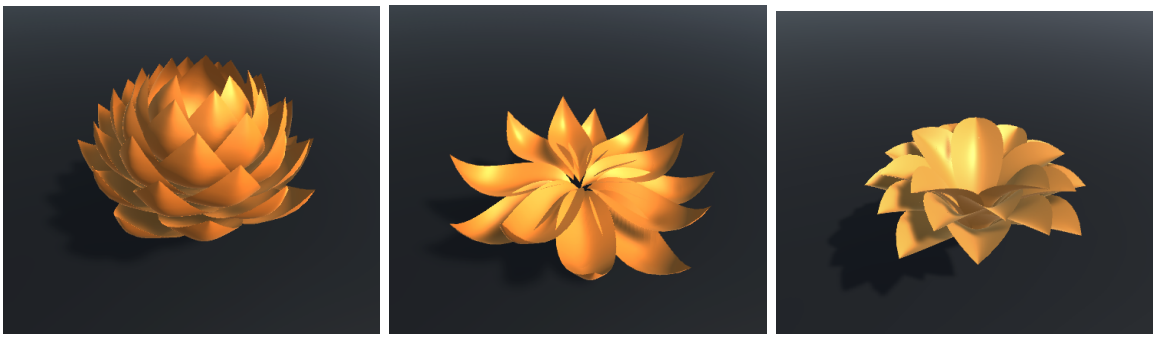
Left: Bicubic patch, control points in white, vertices in blue. Center: Resultant mesh with vertices omitted. Right: Control points arranged to produce a petal.

I played around with the Bezier surface to determine what arrangements of control points produced reasonable-looking petals before writing a procedure to assign randomly-selected coordinates to the control points, thereby generating a random petal. Tuning the procedure to generate realistic-looking petals was a matter of adjusting the constraints that I placed onto the control points and then observing the produced petals. For instance, I constrained opposing vertices across a central axis to always retain bilateral symmetry.

These petals can now be incorporated into the flower generation algorithm, which itself can be modulated by using randomly-selected values for n and c. To further increase the variety of flowers, I added two more parameters to modulate the pitch (upward/downward tilt) of the petal: the pitch rate, which determines the rate at which the petal tilt increases with rising values of n, and the base pitch that the petal tilt is initialized to.

The final procedure is capable of generating a wide variety of realisitic-looking flowers, as demonstrated by the sample flowers below. Still, there is room for improvement that was not explored for the sake of time: further work to improve the fidelity of the flowers might be augmenting the generation routine to repair any mesh clippings, and adding plant organs to the centers of the flowers.





Examples of procedurally generated flowers.

Part 2: Training a Flower Generator

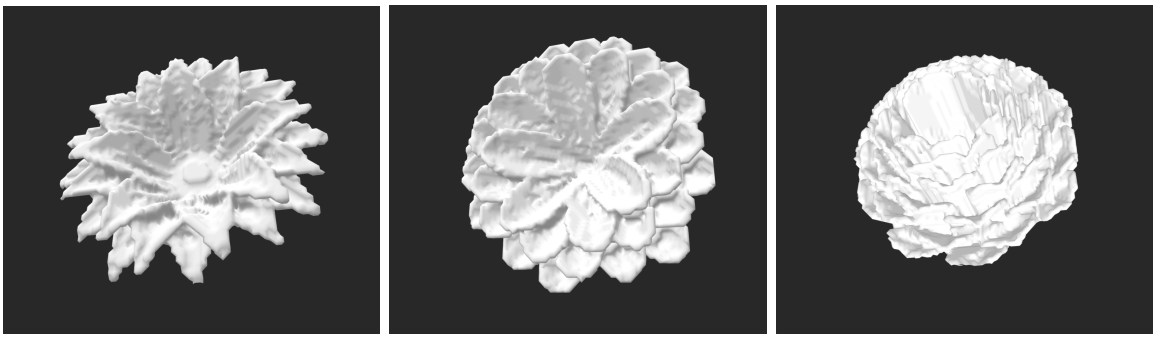
Among the 3D generative architectures that I perused, I found implicit field generators to be the most compelling approach. While architectures such as 3D CNNs and point set generative networks take the approach of having a generator network map directly from a latent space vector to a 3D model (a voxel model and a point cloud model, respectively), an implicit field generator (first described by IM-Net [CZ18]) maps a concatenation of the latent space vector and one of a number of point coordinates into a boolean of whether the point belongs inside or outside of the 3D model. Essentially, the latent space vector maps to a function that implicitly represents the 3D model; this function is referred to as the implicit field.

The benefit of this approach is that in theory, output shapes can be sampled at arbitrarily fine resolutions using the Marching Cubes algorithm (described succinctly here), which extracts a triangle mesh from the implicit field by iteratively placing cubes onto regions of the field, and producing different triangles depending on which of the eight vertices of the cube are inside/outside of the field. This is in contrast to the other approaches, whose outputs are inherently constrained to the resolution that the model is trained to generate. It is because of this benefit, as well as the uniqueness of the approach, that I decided to train an implicit field network to replicate the flower-generating procedure that I had written in Unity.

The implicit field architecture works by first training an autoencoder to convert voxel models into latent space vectors, and then training a generative network such as a GAN to generate vectors that can be converted into new models. Therefore, to train the network, the mesh models must be converted into voxel models. Furthermore, to provide a ground-truth implicit field for training, points are sampled from the surface of and the outer space of the voxel model.

I first generated and downloaded 2500 flowers, and then converted each flower into a 256^3 voxel model using the binvox voxelization algorithm. I found that, compared to other 3D models, my flower models were somewhat challenging to voxelize because the flowers consist solely of thin petals; this meant that the voxelization algorithm sometimes left holes in or ignored petals completely. Luckily, this issue subsided after tweaking the parameters of

the binvox command (especially helpful was first generating 512^3 voxel models and then downsampling by a factor of 2).



Voxelized flowers, after re-extracting the mesh using Marching Cubes.

I then used the preprocessing code from IM-Net to sample points. If points were simply randomly sampled from the entire model, a large number of points would be needed to effectively capture the shape of the object. A clever approach by IM-Net is to concentrate sampling in the space that closely surrounds the object's surface.

In addition, IM-Net found that training more effectively led to convergence when the autoencoder was first trained with points sampled from lower resolutions, and then trained with higher resolutions in later runs. This was done by downsampling the voxel model (a matter of grouping voxels and then mapping them to a single output voxel that is occupied if one member of the group is) and then repeating the point sampling process. So while the voxel models fed to the encoder were always 256^3, the decoder could be trained with one of three resolutions: 16^3, 32^3, or 64^3.

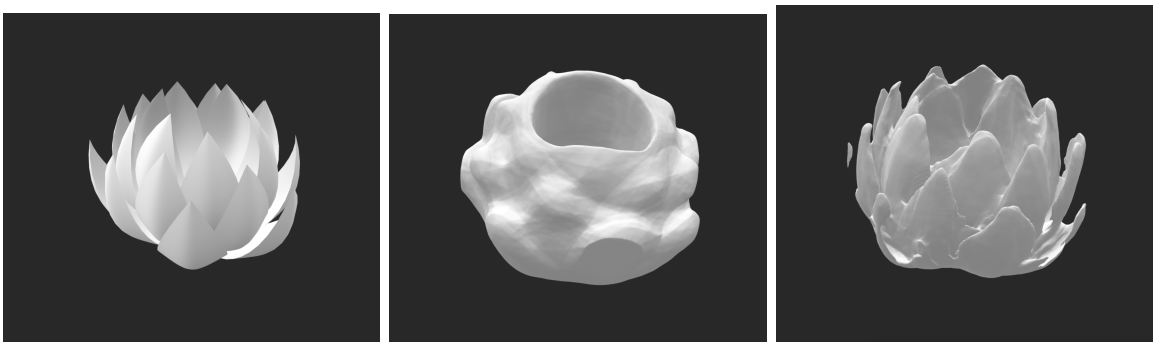
With preprocessing completed, it was now a matter of writing the model architecture and training the model. While my implementation takes heavy pointers from the original implementation of IM-Net, I wrote my own architecture and training loop using Tensorflow 2, which the authors do not provide an implementation in.

The encoder's trainable layers consist solely of 3D convolutional layers that gradually shrink the input voxelized model down into a 256-long latent space vector, with ReLu activations in between and a sigmoid activation at the output. For the sake of improving convergence time and the quality of the output models, I later added batch normalization layers and replaced ReLu with LeakyReLu. The decoder is a simple MLP that also uses LeakyReLu and rounds the output logit to either 0 or 1, corresponding to whether the inputted point belongs inside or outside of the model. Later, I took a cue from the [CZ18] implementation and replaced this rounding with a "leaky" rounding, which produces roughly 0 or 1 but still retains a small portion of the orginal logit:

$$\text{output} = \max(\min(\text{logit}, \text{logit} * 0.01 + 0.99), \text{logit} * 0.01)$$

The loss objective of the autoencoder is simply MSE on the decoder's point values.

I found that my first train-through of the autoencoder led to horribly oversimplified decoder outputs: each outputted shape, regardless of the input, was a very undetailed blob. At the very least, the model was recognizing that all of the input data had some form of rotational symmetry. I found that following the methodology of IM-Net and training starting at a resolution of 16^3 and then moving up to 32^3 and then 64^3 after 200 epochs of training each led to vastly better convergence, although the produced models were still somewhat poor, resembling poached eggs more than flowers. After tweaking the learning rate, weight initialization, and making the architecture changes described above, results were satisfactory, but still far from perfect.

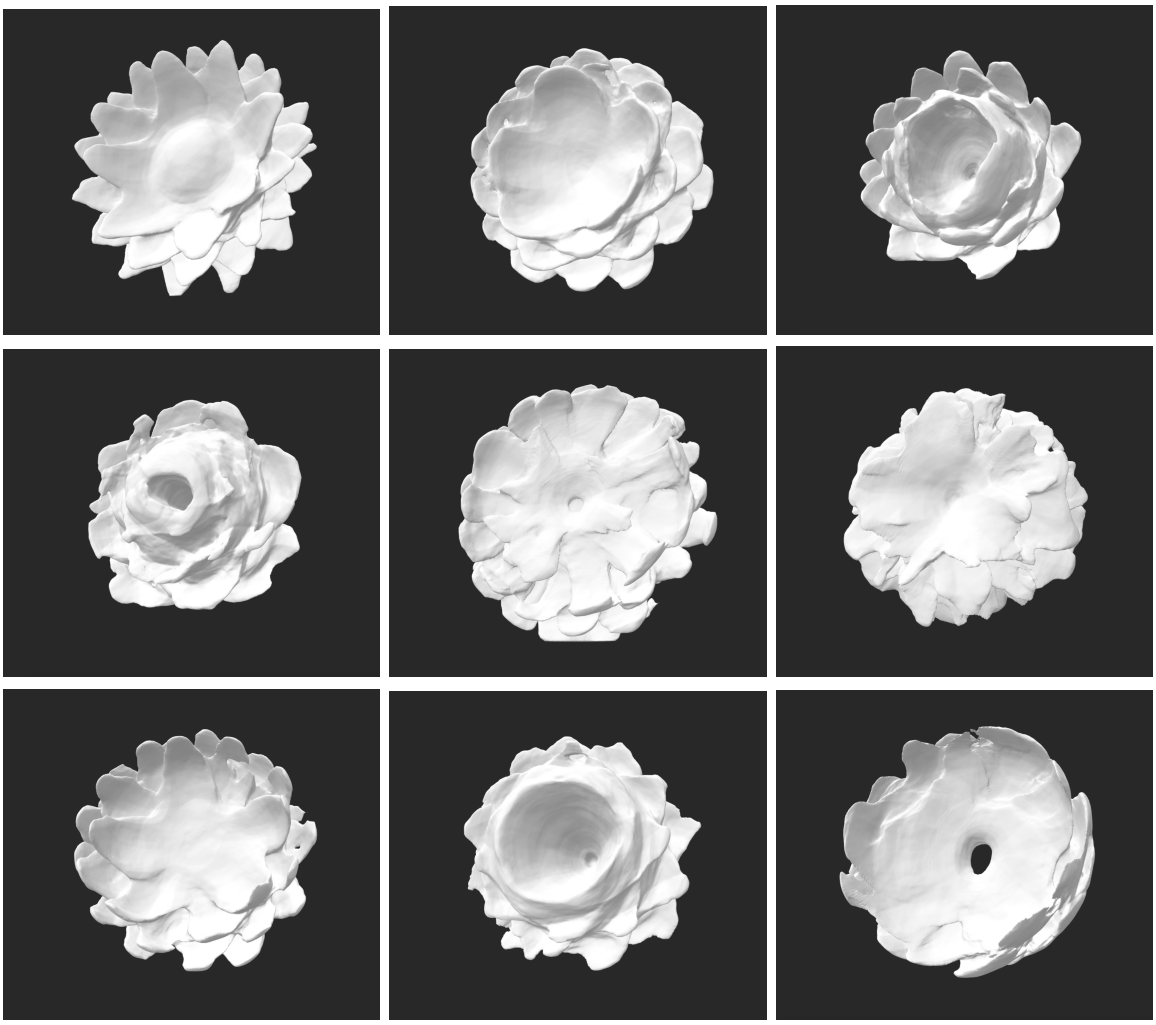


Left: Sample test set flower before voxelization. Middle: Reconstructed flower from autoencoder trained at 16^3 resolution. Right: Reconstructed flower from autoencoder trained at 64^3.

It was finally time to use the autoencoder to generate new flowers by training a GAN to generate new latent space vectors. My first GAN was written based off of one from a CS1470 lab, but from the produced flowers and plots of generator/discriminator loss, I noticed that mode collapse was occurring: the discriminator quickly overtook the generator during training, and the flower models corresponding to the generated latent space vectors all looked virtually identical.

Eventually, I minimized the effects of mode collapse by using a Wasserstein GAN [ACB17], in which the discriminator becomes a critic network whose output is kept as a logit and is treated as a "realness" score. Furthermore, I used gradient penalty [Gul+17] to keep the weights at a low magnitude, which the authors find to be crucial for WGAN's effectiveness.

The generated flowers mostly retain the distinctive feature of containing petals arranged around a central axis. However, the quality of samples isn't always spectacular, suggesting that there is still room for improvement when it comes to the architecture and training of the autoencoder and/or GAN. In particular, the at times the petals appear to mesh together too much rather than appear as separate entities, and the centers of some flowers resemble a bowl or a doughnut. This is likely due to fact that the center of the flower, in which the petals of the flower come together, is the hardest to capture in high detail given the limited resolution of point data.



Sample flowers created from latent vectors generated by the GAN.

Concluding Thoughts

While the generated flowers are not of perfect quality, I was pleasantly surprised at how well the autoencoder/GAN were able to generate objects that even resembled flowers in the first place, especially given that, prior to this project, I had no experience working with 3D data or models that operate on 3D data.

Improvements might arise from further tuning hyperparameters, using a larger batch size for stabler convergnece (although this would require a GPU with more memory than I had access to), or through further enhancements to the voxelization/point sampling processes that are better suited to the thinness of the flower models.

References

[PL96] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. en. The Virtual Laboratory. New York, NY: Springer, Mar. 1996.

[Sur03] Jaylinda Suridge. "Modelling by Numbers". en. In: (Sept. 2003).

[ACB17] Martin Arjovsky, Soumith Chintala, and Léon Bottou. "Wasserstein GAN". In: (2017). eprint: `arXiv:1701.07875`.

[Gul+17] Ishaan Gulrajani et al. "Improved Training of Wasserstein GANs". In: (2017). eprint: `arXiv:1704.00028`.

[CZ18] Zhiqin Chen and Hao Zhang. "Learning Implicit Fields for Generative Shape Modeling". In: (2018). eprint: `arXiv:1812.02822`.