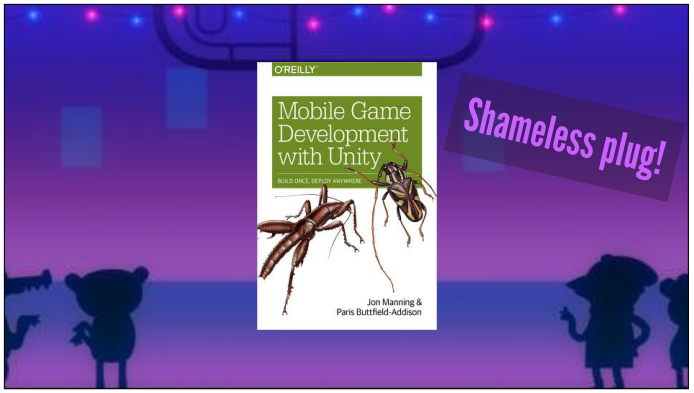
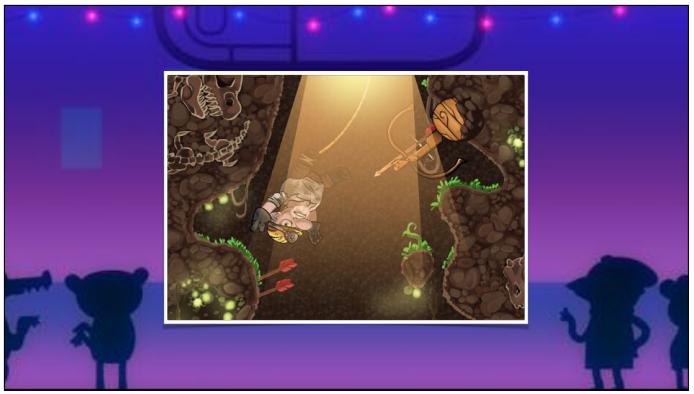
I work for Secret Lab, development studio based in Hobart in Australia

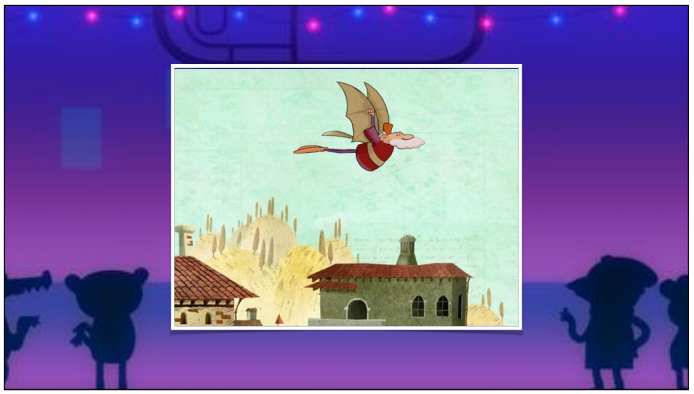


Written about 20 books, this is our most recent one



Working on a game called Gnome's Well that Ends Well in which you dangle a gnome on a rope down a well

---

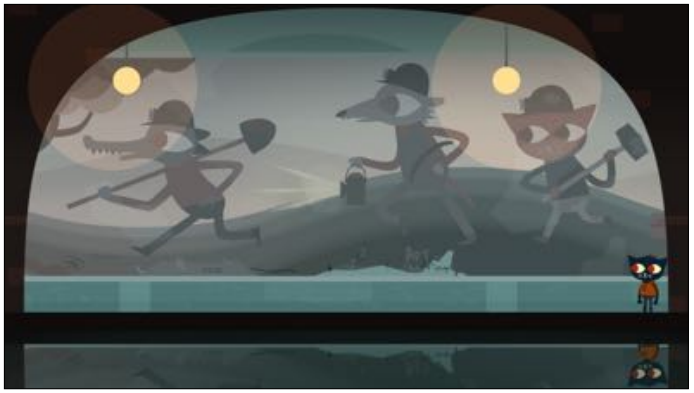


And a game called Leonardo's Moon Ship, which is a beautiful game about the life and times of Leonardo da Vinci!

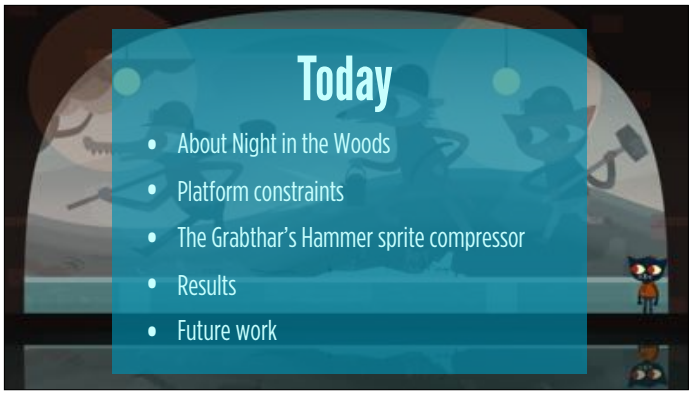
Recently I worked with Infinite Fall and Finji…

…on a game called Night in the Woods

We’re porting Night in the Woods to iOS Hooray!

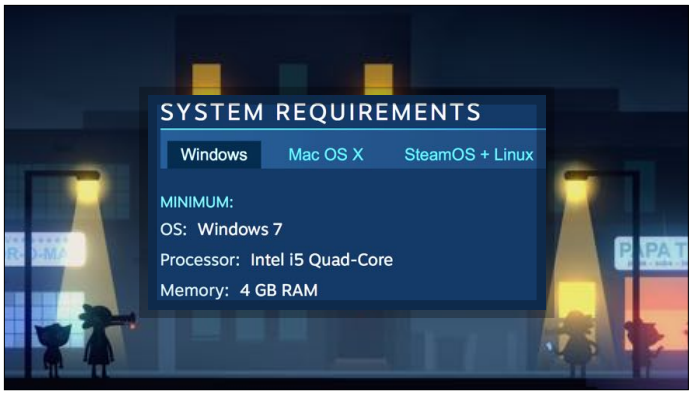


Let’s talk about where we’re going.



Mae has dropped out of college and returned to her hometown, where she finds that while the rest of her friends have grown up and are moving on with their lives, she really hasn’t.

It’s got a very visually rich world, which is all done with sprites - very few vector based approaches used here, for production reasons.

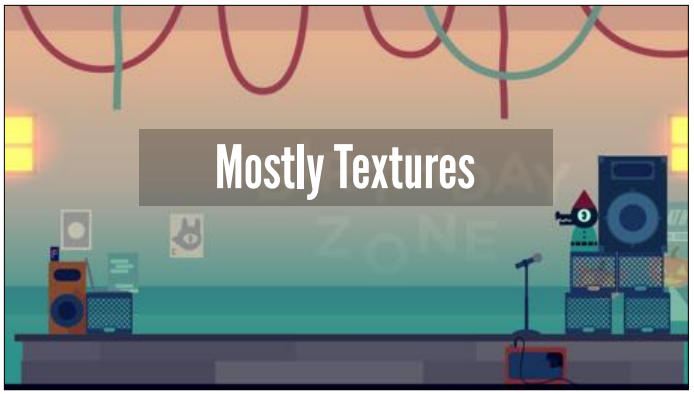


System requirements are 4GB of ram

We do get up to 4GB of memory on higher end devices, but the lowest end is 1GB

Just to get technical on you for a second, here's Angus…

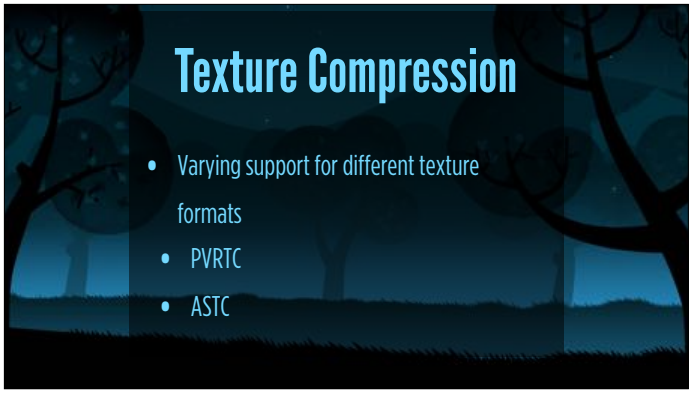
So, the main challenge facing a mobile port of this game is memory.



So, it's important to find ways to reduce the amount of texture memory,

---

We generally use texture compression everywhere we can, but we're limited. PVRTC is available everywhere, while ASTC is available on more modern devices. ASTC is better, providing better compression ratios, image result, and more configurability, but there's no way to ship a product _only_ to ASTC-supporting devices on the App Store (because certain A7-powered devices run iOS 11.)



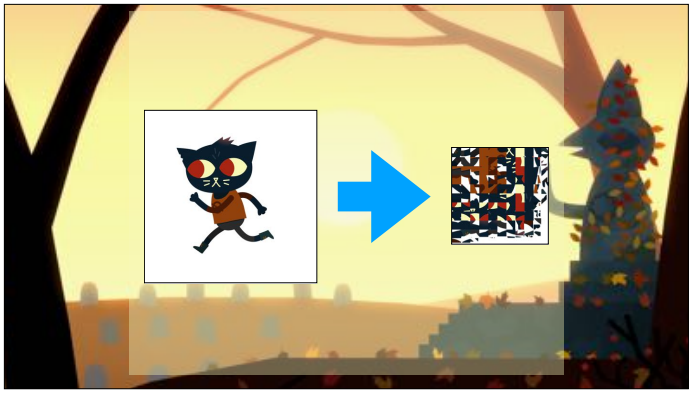
The compressed texture formats available to us put some interesting constraints on us

---

And then there's the technique we used

The general approach

So, we use sprite dicing heavily, and this is what I'll be talking about with you today.

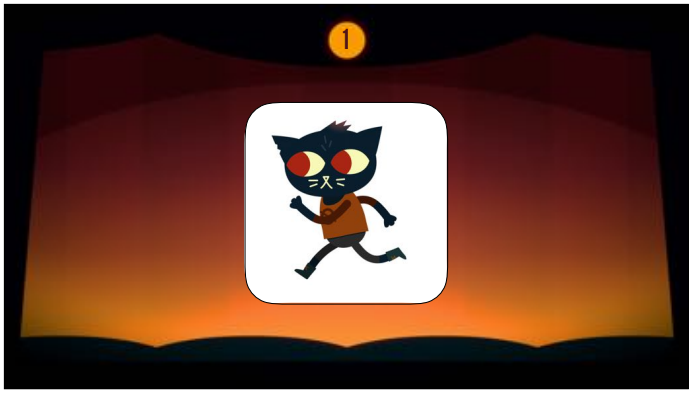


Here’s a peek at the result of the process outlined in this talk. We’re taking a large sprite that has lots of empty space around the edges, but also within its trim-able area. It also has quite a bit of redundant areas of the same colour.

The original is 720x720; the resulting pieces are able to be packed into an area less than 512x512. To render the image, we create a mesh that samples the appropriate parts of the texture.

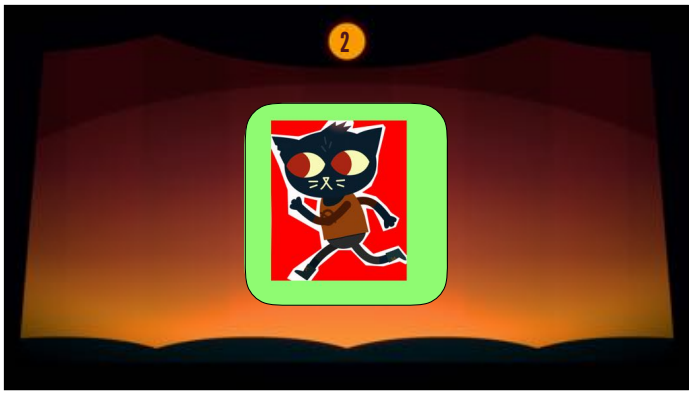
This is all existing stuff; what I’m about to start talking about is what we built on top of this technique.

The core idea of sprite dicing is not brand new.
“Chopper” was a sprite compression tool used in Aladdin (1993) and other Virgin Media games
Originally developed by Andy Astor, worked on by others
Divided frames into smaller tiles, removing transparent regions
Frames are reassembled at runtime
Also stored stuff like per-frame hit box information



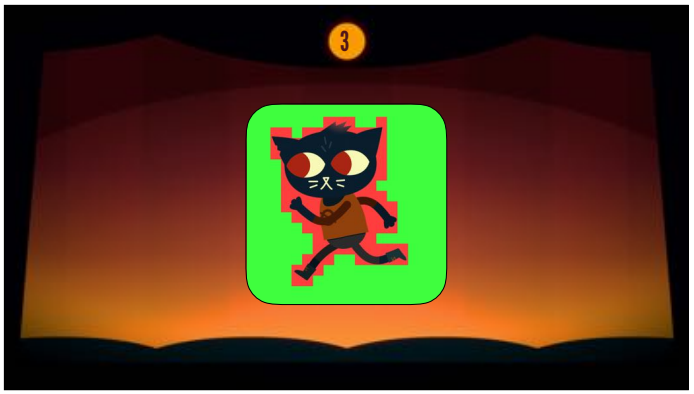
If a sprite is concave, it contains lots of transparent pixels that a simple trim can't remove

---

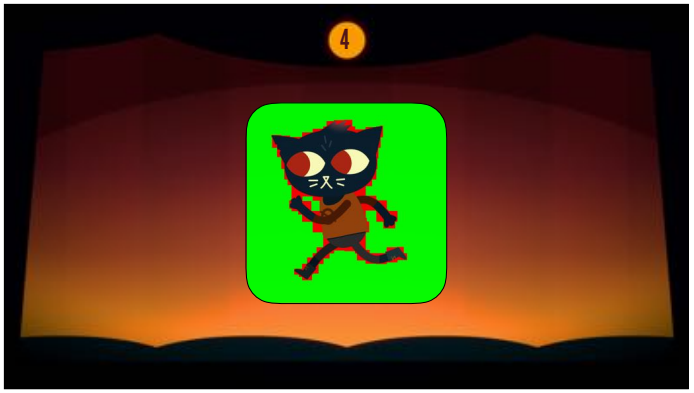


The green area here is what we could get rid of with a simple trim; the red area is stuff that’s transparent, but isn’t reachable because it’s within the trim box

---



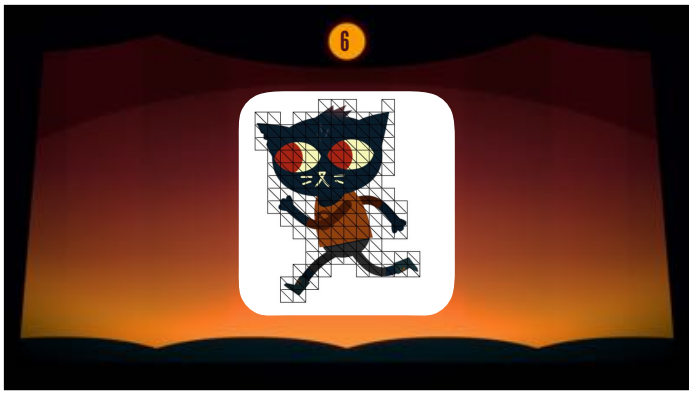
Everything here marked green is stuff we can discard.
Note that there’s some unnecessary transparent area here that we can discard - this is because these pieces are 32x32, BUT:



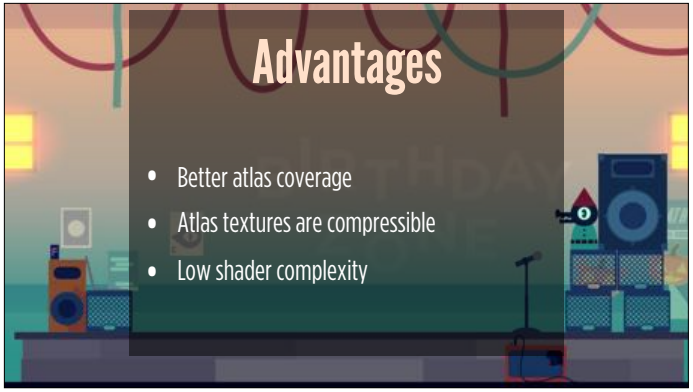
There’s no reason for each piece to be 16x16, so we trim those to the smallest individual rectangle as well.

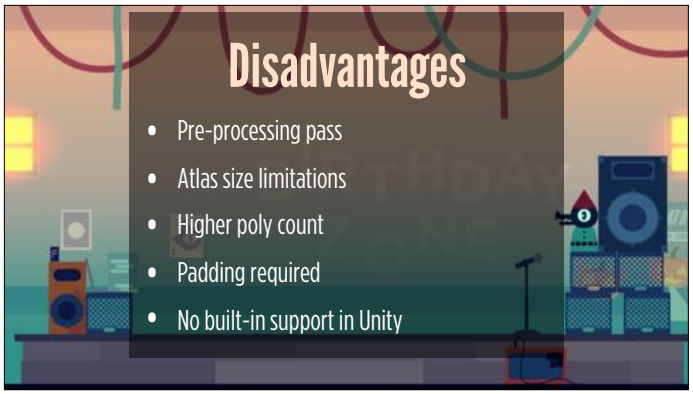
this is a 720x720 image, with all of the transparent pieces removed, the pieces individually trimmed, and packed into a sheet - it fits into a 512x512, with lots to spare.

It has to be 512x512 because PVRTC requires a power-of-two texture.



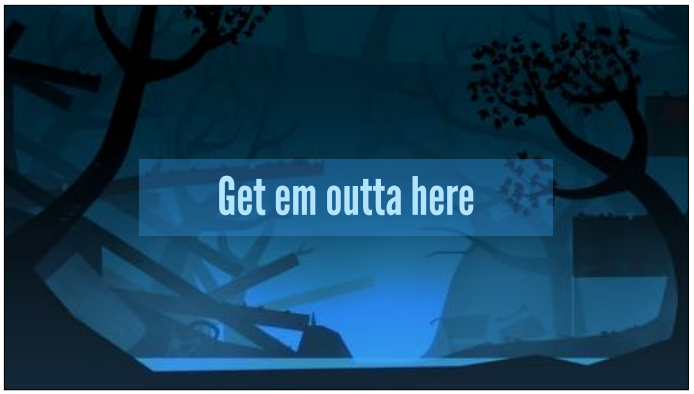
The downside to this is that our mesh density has gone up a bit - each vert is like 18 bytes. However, the additional mesh cost is much lower than the savings we get by storing fewer pixels of texture data.

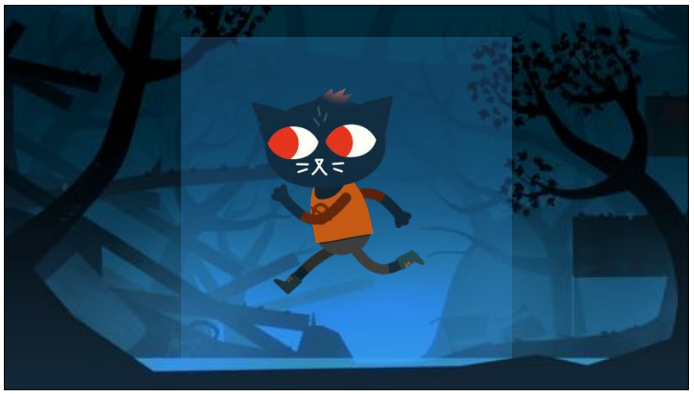




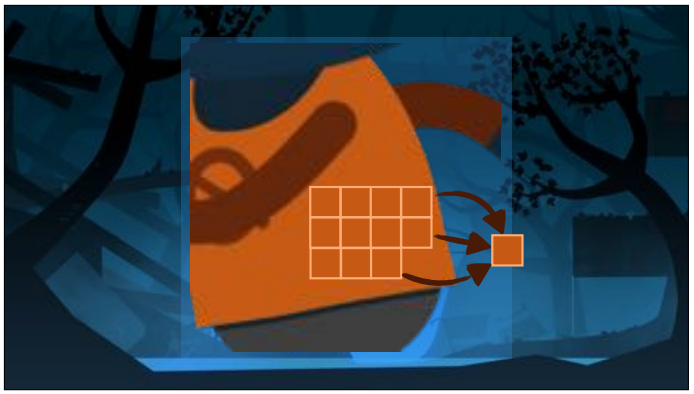
It’s a compression tool, which produces significant savings.


Find bits we don’t need


Get em outta here



These blocks are all the exact same pixels.

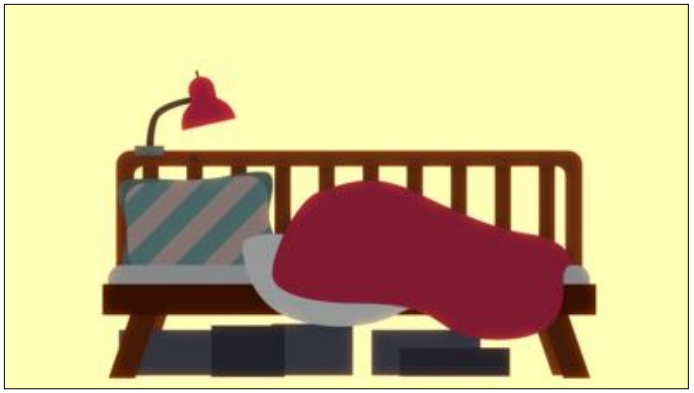


There’s no reason for all of these pixels need to be stored, when they’re all copies of another.

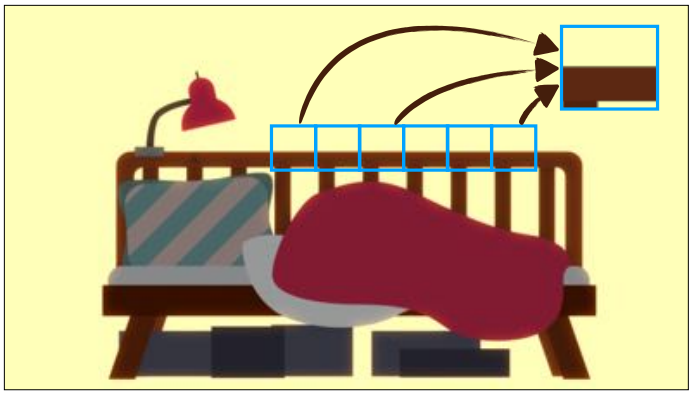
Not just for flat areas!

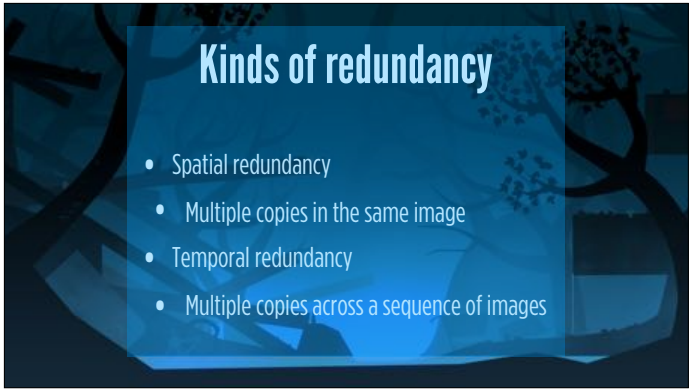
Remove mae, she’s just a head and an arm in this scene

Get rid of the background

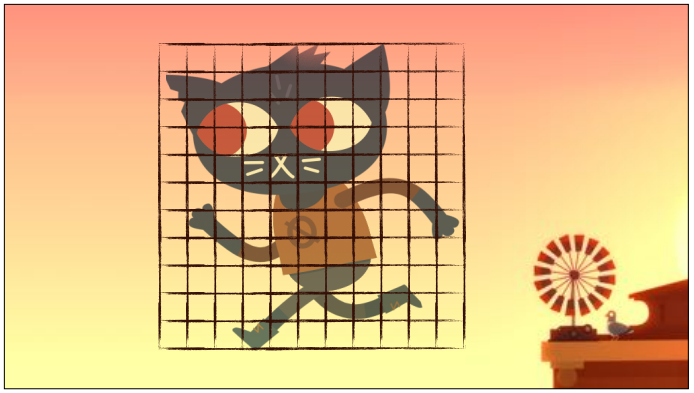


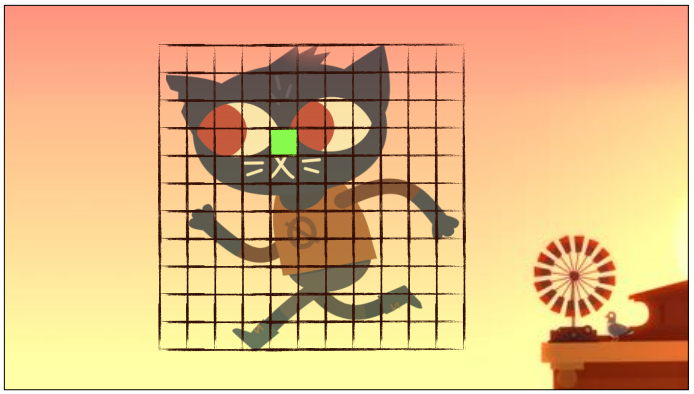
Turn off that bloom



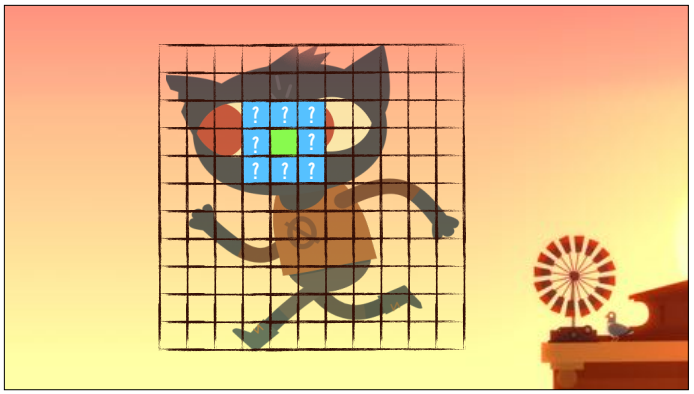

Kinds of redundancy
Spatial redundancy
Multiple copies in the same image
Temporal redundancy
Multiple copies across a sequence of images


Finding spatial redundancy

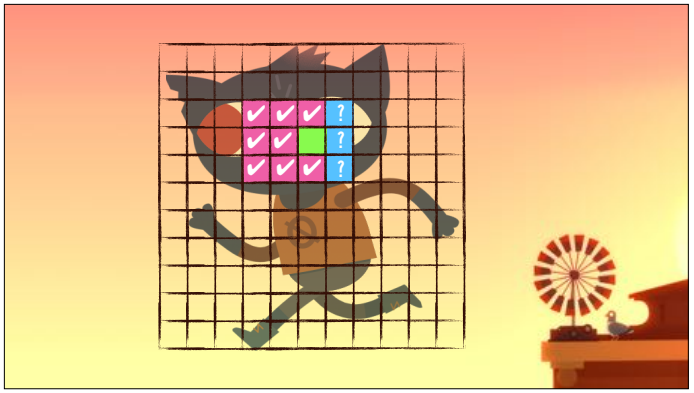




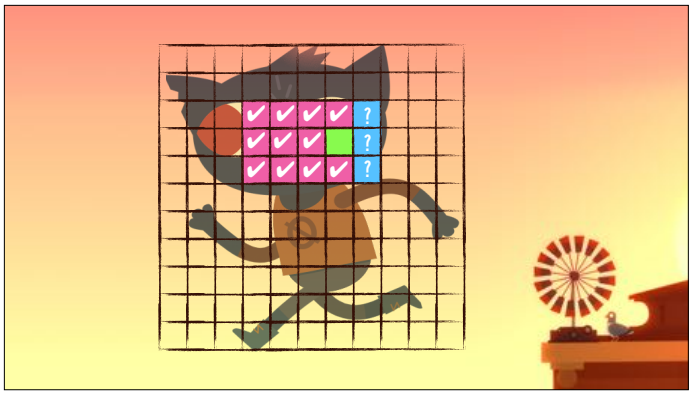
We exploit the fact that it’s very unlikely to have two pieces that are identical being very far apart. So, we only perform tests on pairs of pieces that are close together.

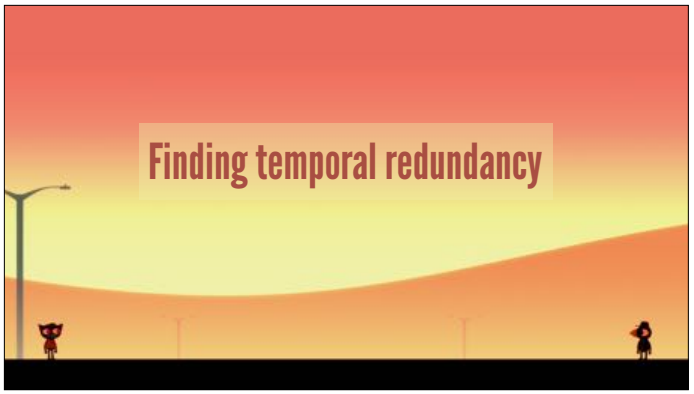


Are they the same?

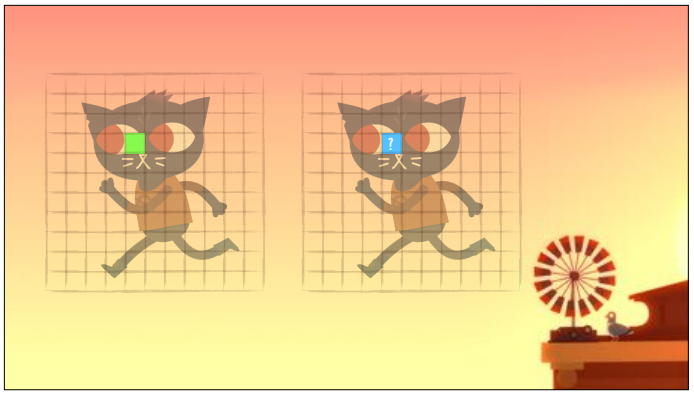


Only unique pairs are checked



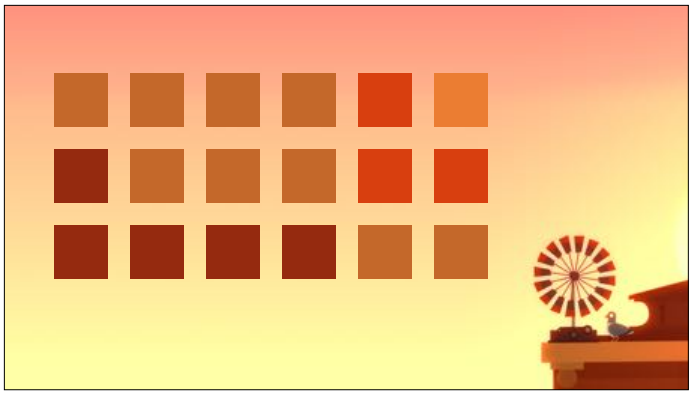


Here’s Mae’s run cycle

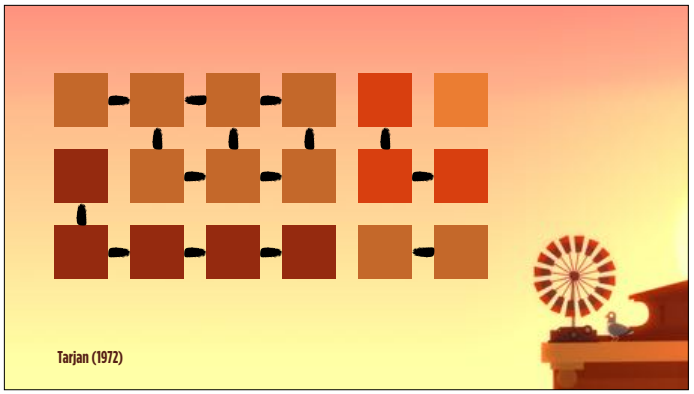


We'll check the chunk at the same position in both images; repeat for all chunk positions, and for all pairs of images (or, if we want to be faster, between pairs of images that follow each other in the sequence)

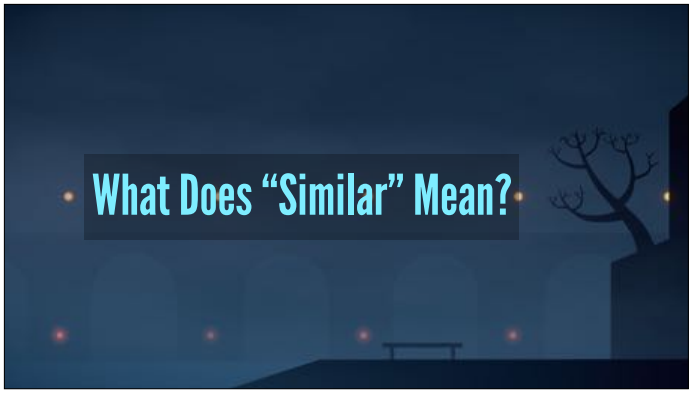
At this point we've built up an idea of how similar different chunks in the image are to each other, and we want to find clusters of similar chunks (so that we can pick one and keep it)



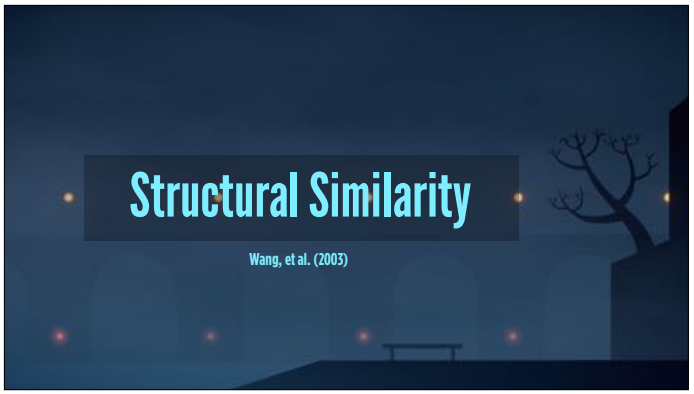
We now have a graph of chunks…



…and their connections - these connections can be between chunks on the same image, or chunks on multiple images.
We consider a chunk to be 'connected' to another if their similarity score is high enough
We can then find the "strongly connected sets" of chunks; in each set, all chunks are going to be very similar to each other, so we pick any of them and make the rest use that chunk's texture data
We use Tarjan's algorithm to find the strongly connected sets


What Does “Similar” Mean?

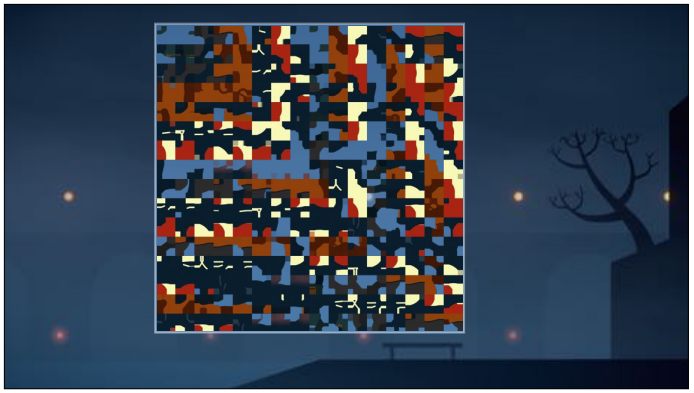

Not “the same pixels”


Structural Similarity
Wang, et al. (2003)


Compares the shapes in two images


Keep only the unique parts


Packing the Atlas

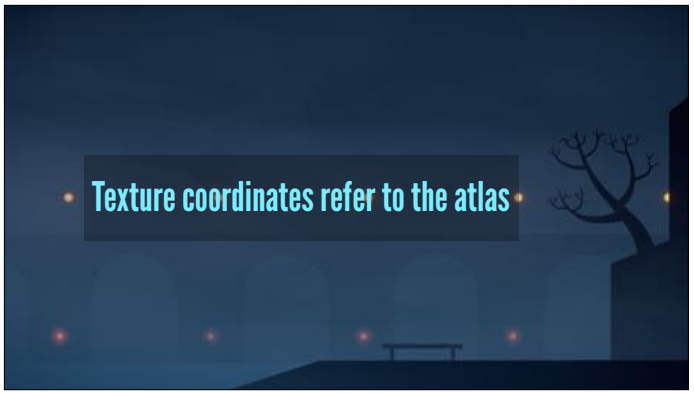


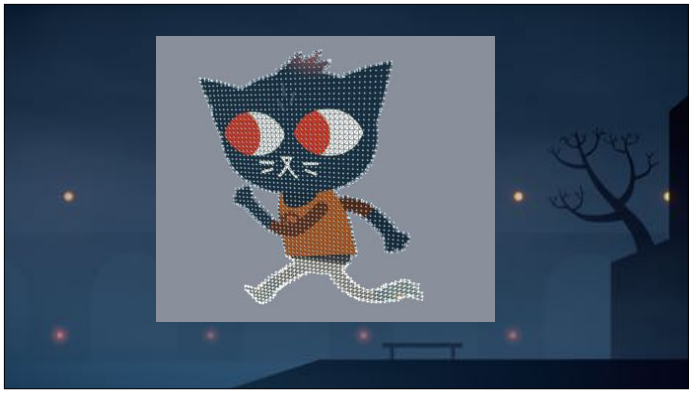
Basically have to dilate each chunk to prevent bleeding

This is wasted space since these pixels are only sampled at low mip levels


Generating the mesh


Each chunk becomes triangles


Texture coordinates refer to the atlas

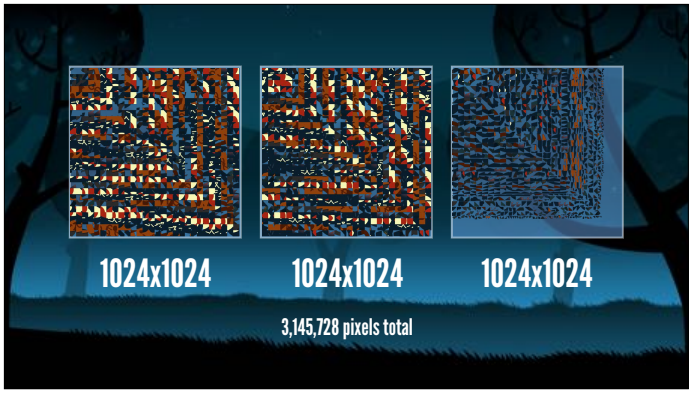


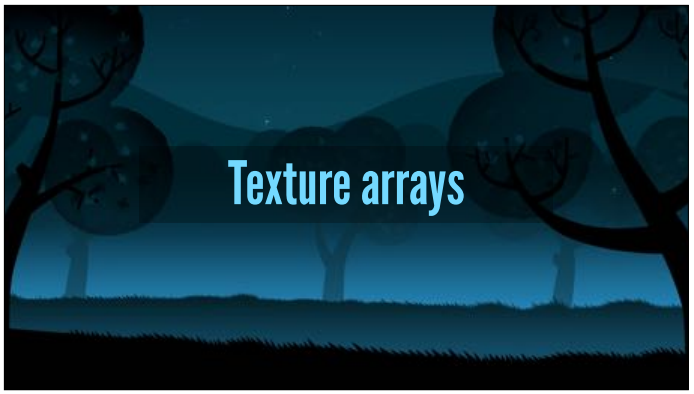
Conforms more closely to the outline of the character; lots of texture reuse in this mesh; more vertices

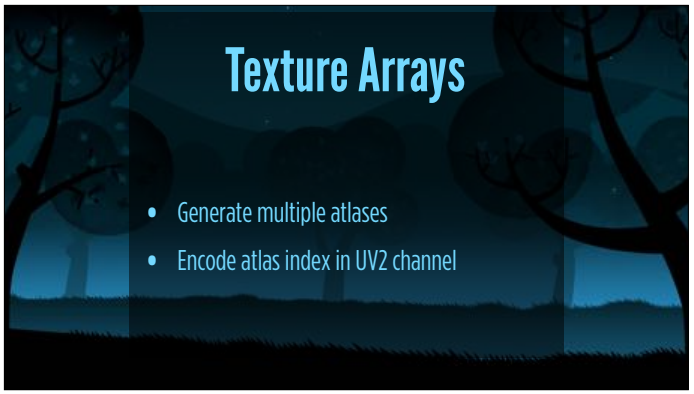
---

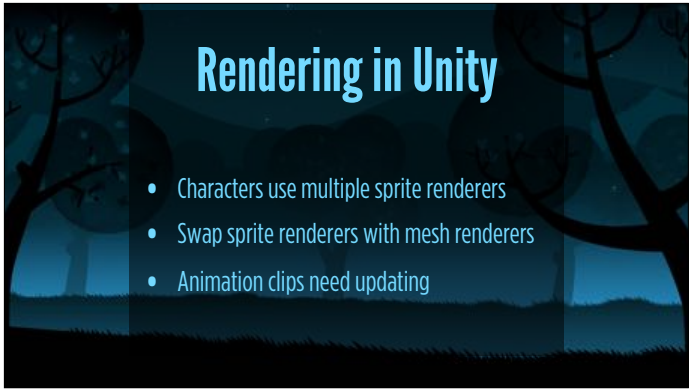
Max limit is 8192 on iOS

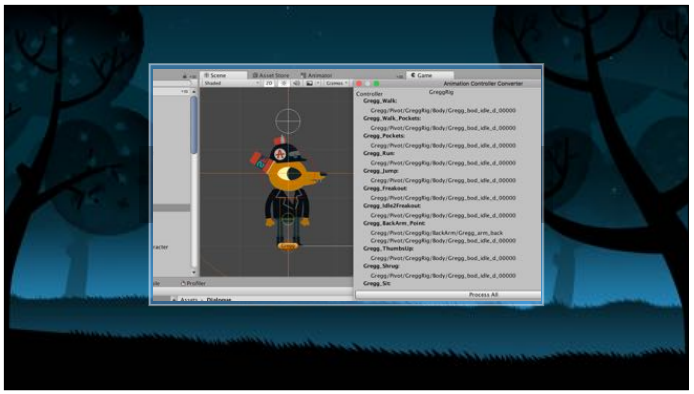
Notice the empty area at the right and bottom edges - that's wasted space


1024x1024
1024x1024
1024x1024
3,145,728 pixels total

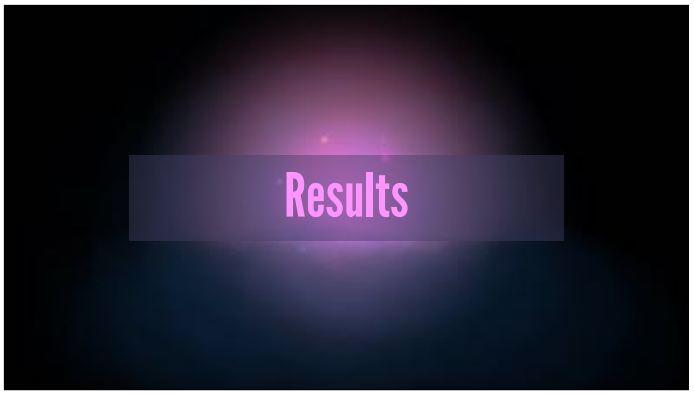

Texture arrays

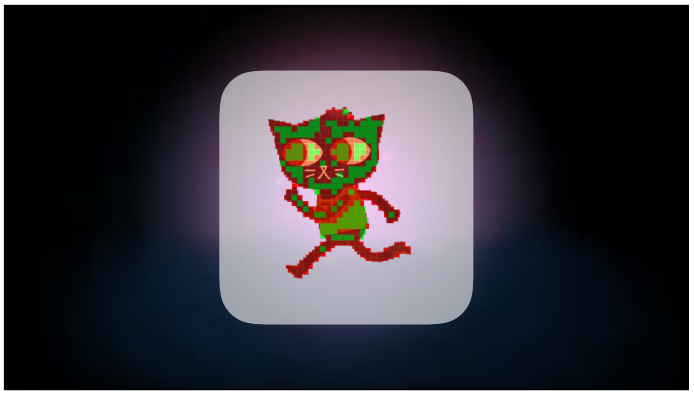

Texture Arrays
• Generate multiple atlases
• Encode atlas index in UV2 channel





Built tools to automate converting entire rigs of characters, swapping out sprite renderers with mesh renderers, replacing animation clips that change sprites to ones that change meshes





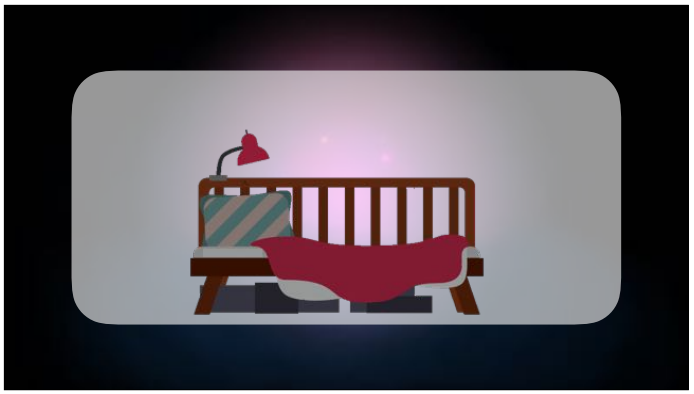
All blocks that have a green tint are re-used copies of something that exists elsewhere in the atlas. Red-tinted blocks are blocks that exist in the atlas.

Note how the green blocks are largely flat areas of colour, but non-flat blocks exist, such as near the waistline of Mae's shirt - it's green because another frame elsewhere in the sequence has a copy of the pixels, so only one copy is needed.

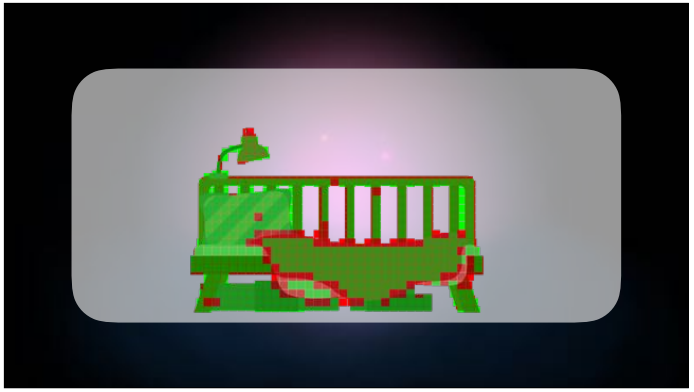
The 'original, trimmed' number is actually being generous, because it assumes PVRTC can work with NPOT textures, while the 'diced' number does not include that assumption

This is actually one of our worst cases, because there's so much movement in between frames and not much redundancy

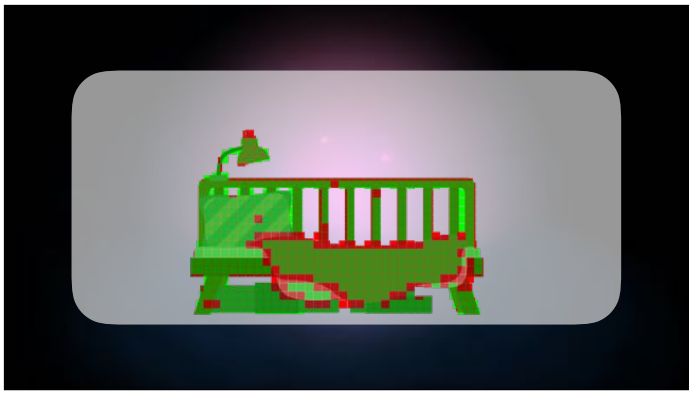
---



We get even better performance if there are non-moving parts of an image. In this example, only the blanket's moving.



Notice how mainly the edges of the blanket are unique, while the flat interior of the blanket is de-duplicated along with the pillow, bed frame, and boxes under the bed, which never move.

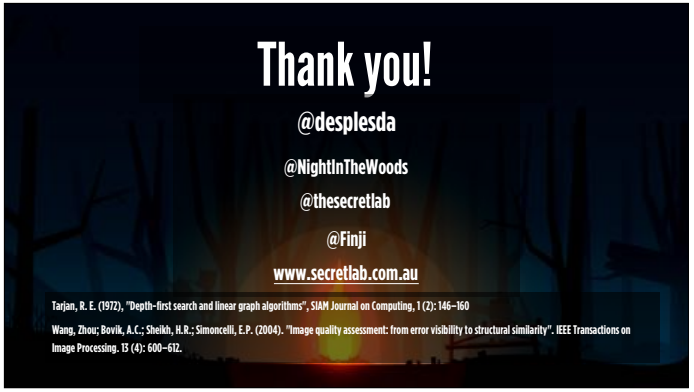


The ‘original, trimmed’ number is actually being generous, because it assumes PVRTC can work with NPOT textures, while the ‘diced’ number does not include that assumption

This is actually one of our worst cases, because there’s so much movement in between frames and not much redundancy

Here’s what this looks like in Unity. We have to use a mesh renderer to do these, since the built-in sprite renderer doesn’t let us override the UV coordinates of the generated mesh.

Looking out to the future


Thank you!
@desplesda
@NightInTheWoods
@thesecretlab
@Finji
www.secretlab.com.au
Tarjan, R. E. (1972), "Depth-first search and linear graph algorithms", SIAM Journal on Computing, 1 (2): 146–160
Wang, Zhou; Bovik, A.C.; Sheikh, H.R.; Simoncelli, E.P. (2004). "Image quality assessment: from error visibility to structural similarity". IEEE Transactions on Image Processing. 13 (4): 600–612.