General Information

SpriteSharp is an efficient solution for generating better meshes for sprites, helping reduce your 2D scene total polygon count, draw call amount, overdraw and CPU load.

While Unity itself is capable of doing this, the possibilities are limited. There are no options to tweak to get the best possible performance, and sometimes meshes generated by Unity are so suboptimal you have to revert back to using *Full Rect* quads, losing the overdraw improvement.

This is why *SpriteSharp* was developed. It extends built-in Unity sprite mesh generation with various tweakable options, allowing you to get the best performance out of your 2D game. It is especially useful when working on complex scenes and developing for mobile platforms, where overdraw is often a problem.

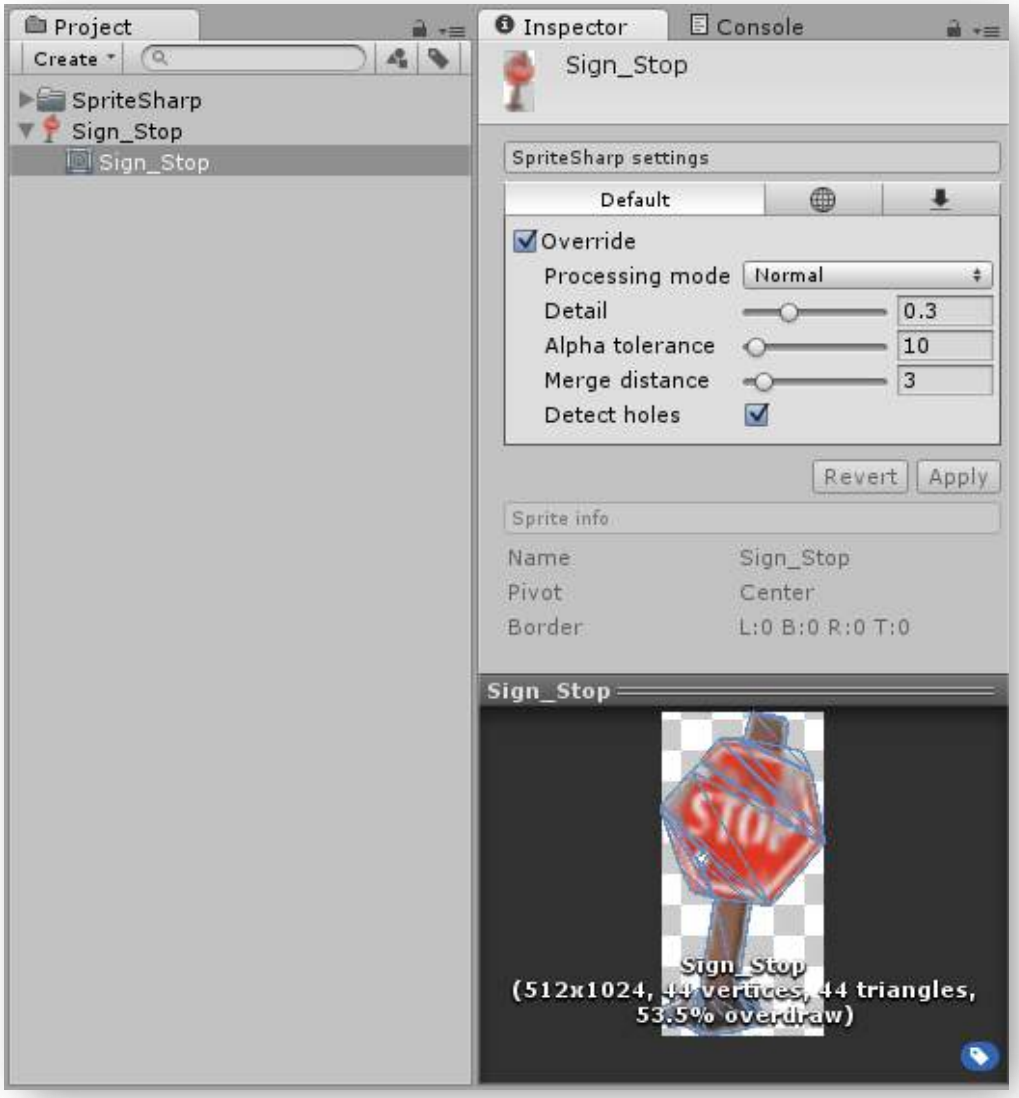
Just a few more points:

- If you are drawing a lot of sprites, the difference between 1000 and 100 triangles per sprite is going to end up a big deal.
- Objects with `SpriteRenderer` are only batched up to a certain amount of triangles, and the limit of triangles per batch is around 700. This is because sprites are batched on the CPU, and doing that has a slight overhead. Therefore, decreasing the polygon count also decreases the amount of draw calls and offloads the CPU for more interesting work, like running your game scripts.
- Unity has no prior knowledge about how sprites will be scaled in the scene. There is no point of making highly-tesselated mesh for an object that takes a small area on the screen. That's why manual control is needed.

Unity 5.1 and newer is supported, both Personal and Pro.

Overview

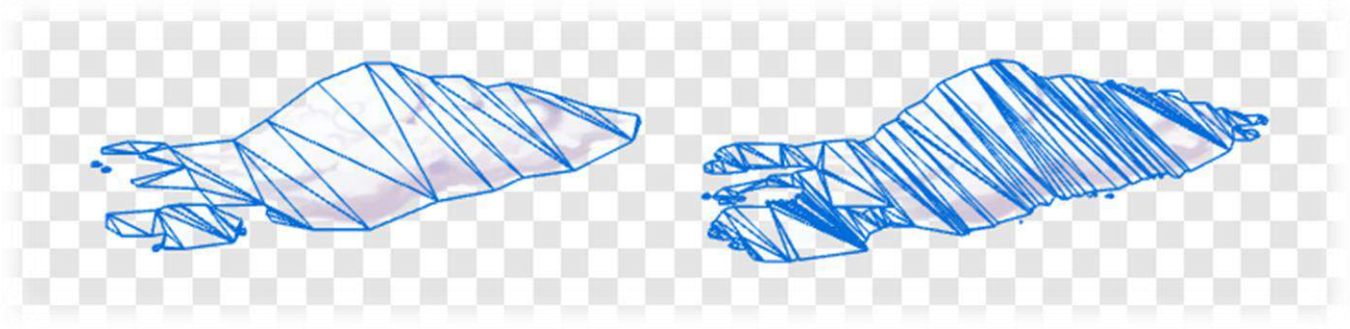
SpriteSharp integrates seamlessly into Unity and is very easy to use. Just make sure you have `Mesh Type` set to "`Tight`" in the texture settings, and select the sprite in your `Project` view. *SpriteSharp* interface will show up in the `Inspector`.



SpriteSharp has four different method of sprite processing. They have different use cases, and sometimes it's a good idea to experiment to find out what works best for your specific situation. Sometimes *Precise* method may work better than *Normal* even on a complex sprite. Let's take a closer look at the processing methods.

Normal

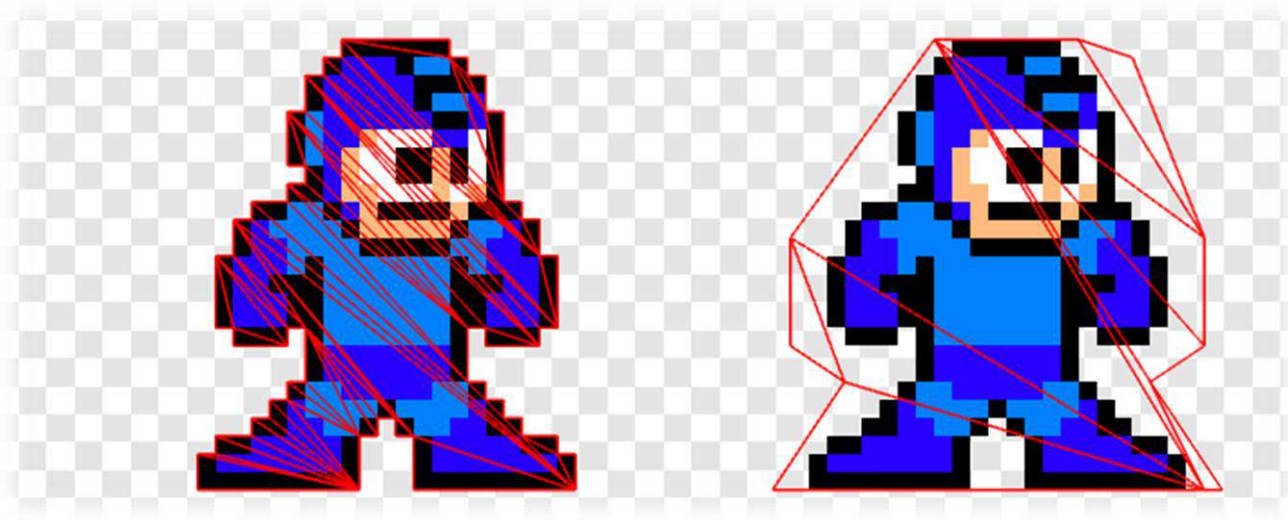
This method generally produces good meshes and has a lot of settings to tweak. It tries to track sprite contours, and is most similar to the built-in Unity algorithm. This is the method you'll use most frequently.



Same cloud sprite with different Normal method settings.
Note how the image on the right has far more polygons, even though overdraw is almost the same.

Precise

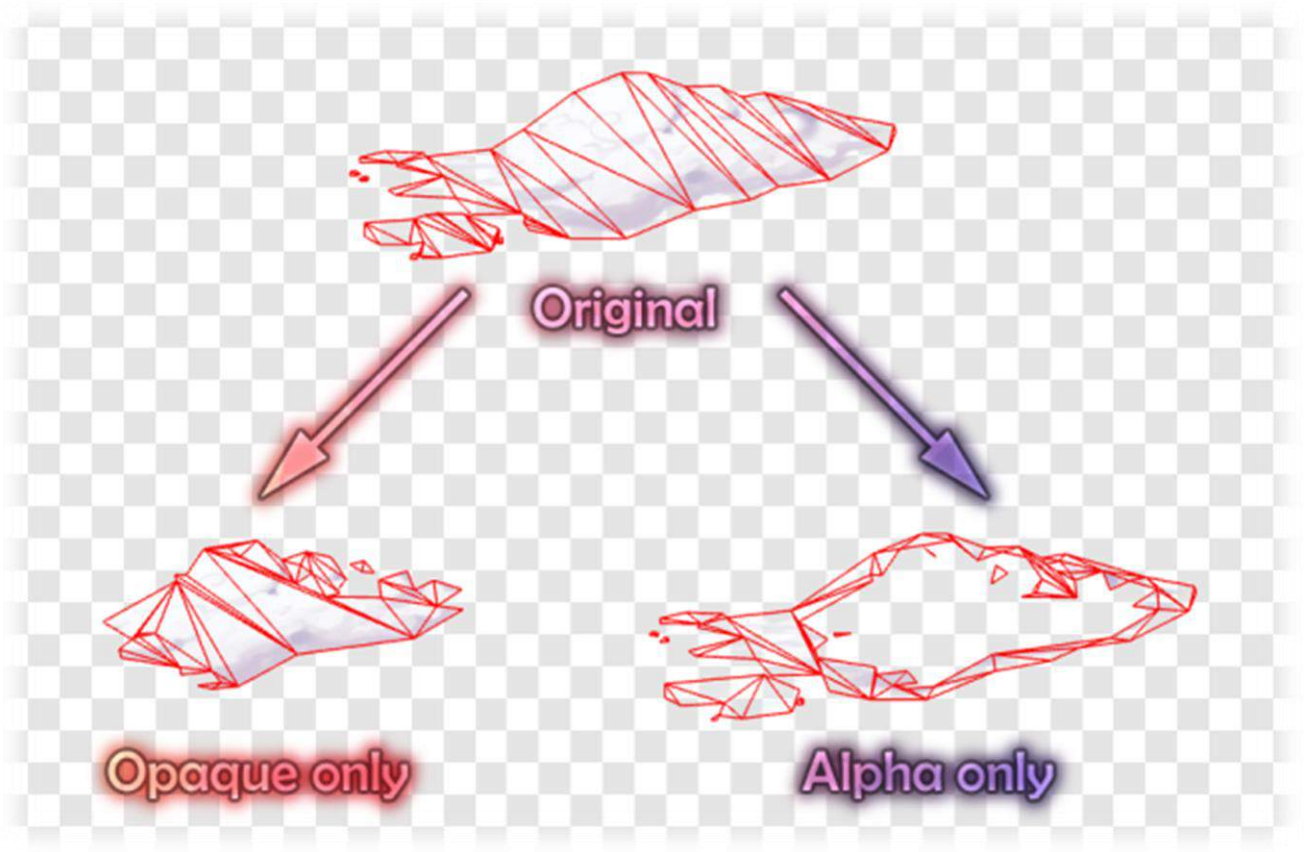
With this method, the sprite is analyzed exactly, pixel-by-pixel, creating precise and tight meshes, independent of texture resolution. This is especially useful for pixel-art sprites, as they often have hard edges and are of small pixel dimensions. It also works great in many other complex cases.



Same character sprite with different Precise method settings.
Note how the image on the left has a "tight" mesh that fits the sprite perfectly, but image on the right has much less polygons.

Alpha Separation

This is the most complex method, and it is designed for sprites that are *mostly* opaque, but have important parts with alpha. The algorithm is identical to *Normal* method, but the resulting sprite is separated into two sprites, with first one containing only 100% opaque pixels, and the second one containing the rest of pixels. These two sprites match up perfectly when placed on top of each other, so the result looks exactly like the original single sprite.



The “opaque-only” sprite is rendered with a simplified opaque shader, with blending disabled and Z-Write enabled. This way, the hardware Z-Buffer testing is utilized, reducing overdraw to *zero* for the opaque sprite part. The “alpha-only” sprite is rendered as usual, with full alpha-blending.

Keep in mind that not all GPUs are doing the Z-Test in the same way, so the actual performance gains may differ. GPUs utilizing *Tile Based Deferred Rendering* (like PowerVR GPUs that are used on all iOS devices) benefit from this the most — rendering overlapped opaque polygons is basically free.

Rectangle Grid

This is not really a method designed to help you optimize performance; instead, it is supposed to be used with custom vertex shaders that modify vertex positions to achieve different kinds of “distortion” effects (like waving flags).

Settings

Similarly to the Texture importer, *SpriteSharp* allows you to set different settings for each build platform.

Let’s take a closer view at settings for each processing method.

OVERRIDE

Enabled and disables *SpriteSharp* mesh generation. Default project settings (which are set in Unity *Preferences* window) will be used when this is turned off.

Normal

DETAIL

Controls how detailed the sprite mesh must be. Lower values results in less triangles, but more overdraw, high values result in low overdraw, but at the cost of more triangles.

This is the value you’d want to tweak first to get a good balance of triangle count and overdraw. A value of 0.3 is a good starting point.

ALPHA TOLERANCE

Pixels with alpha less than this value will be ignored and not included into the sprite mesh. Value of 0 means all non-transparent pixels are included, and value of 254 means only fully opaque pixels will end up in the sprite mesh.

Value of 10-20 generally works well for sprites that are mostly opaque (characters, walls, etc.) without producing any artifacts. It is recommended to keep this value low for sprites representing special effects (smoke, explosions, etc.).

MERGE DISTANCE

Merging vertices that are very close to each other is efficient for decreasing the mesh complexity without sacrificing detail. Controls the maximum distance at which vertices are merged.

Value of 3 works well for most sprites without producing any artifacts. Raising this value too high may result in parts of sprites being slightly cut off.

DETECT HOLES

Controls whether inner holes in the texture will be detected and excluded from the sprite mesh, reducing the amount of overdraw.

It is recommended to keep this option enabled, since it pretty much has no drawbacks. However, you may want to disable it in some special cases.

Precise

ALPHA TOLERANCE

Pixels with alpha less than this value will be ignored and not included into the sprite mesh. Value of 0 means all non-transparent pixels are included, and value of 254 means only fully opaque pixels will end up in the sprite mesh.

Value of 10-20 generally works well for sprites that are mostly opaque (characters, walls, etc.) without producing any artifacts. It is recommended to keep this value low for sprites representing special effects (smoke, explosions, etc.).

EDGE INFLATION

Controls the amount of pixels the sprite mesh will be expanded outwards. Bigger values result in fewer polygons in the generated mesh, but also increase the overdraw.

Value of 0 results in pixel-perfect meshes, which can be used in pixel-art games. In fact, if your pixel-art sprites contain no intermediate alpha values, you can use the included Sprites-Opaque to eliminate *any* overdraw.

Alpha Separation

Most algorithm options are the same as in *Normal.* Sprite will be processed with *Normal* method if no alpha sprite is linked.

LINK/UNLINK ALPHA SPRITE

Separates a sprite into two sprites, first one containing only opaque pixels, and second one containing all other pixels.

INSTANTIATE SPRITES

Instantiates a combined alpha separated sprite. The "alpha-only" sprite is instantiated as a child of "opaque-only" sprite.

To easily attach alpha sprites on existing objects in bulk, use the menu item:

Tools → Lost Polygon → SpriteSharp → Attach Alpha Sprites To Selection

Opaque Sprite Specific Properties:

EDGE CONTRACTION

Controls the amount of pixels the sprite mesh will be contracted inwards. Bigger values reduce the area of the "opaque-only" sprite, but may reduce possible artifacts that can occur on the sprite edge due to filtering.

REDUCE ALPHA BLEED

Controls whether the algorithm will attempt to avoid opaque sprite pixels bleeding into non-opaque pixels. This option is useful for sprites with thin features and small holes, especially when Edge Contraction or Vertex Merge are used with high values.

"Opaque-only" sprite will use the included Sprites-Opaque shader, which a modified version of standard sprite shader with blending disabled and Z-Write enabled. The Sprites-Opaque sprite and material are located at

`Assets/SpriteSharp/Resources/`

Rectangle Grid

Most algorithm settings options are the same as in *Normal.*

X SUBDVISIONS / Y SUBDIVISIONS

Controls the amount of subdivisions of the sprite along the horizontal and vertical axis.

REMOVE EMPTY CELLS

Controls whether to discard cells that have no pixels with alpha higher than Alpha Tolerance.

CULL BY BOUNDING BOX

If this checkbox is set, the tightest possible bounding rectangle will be calculated, and subdivisions will be done within that rect. This is usually good, since it allows to get higher mesh resolution without actually increasing the number of subdivisions.

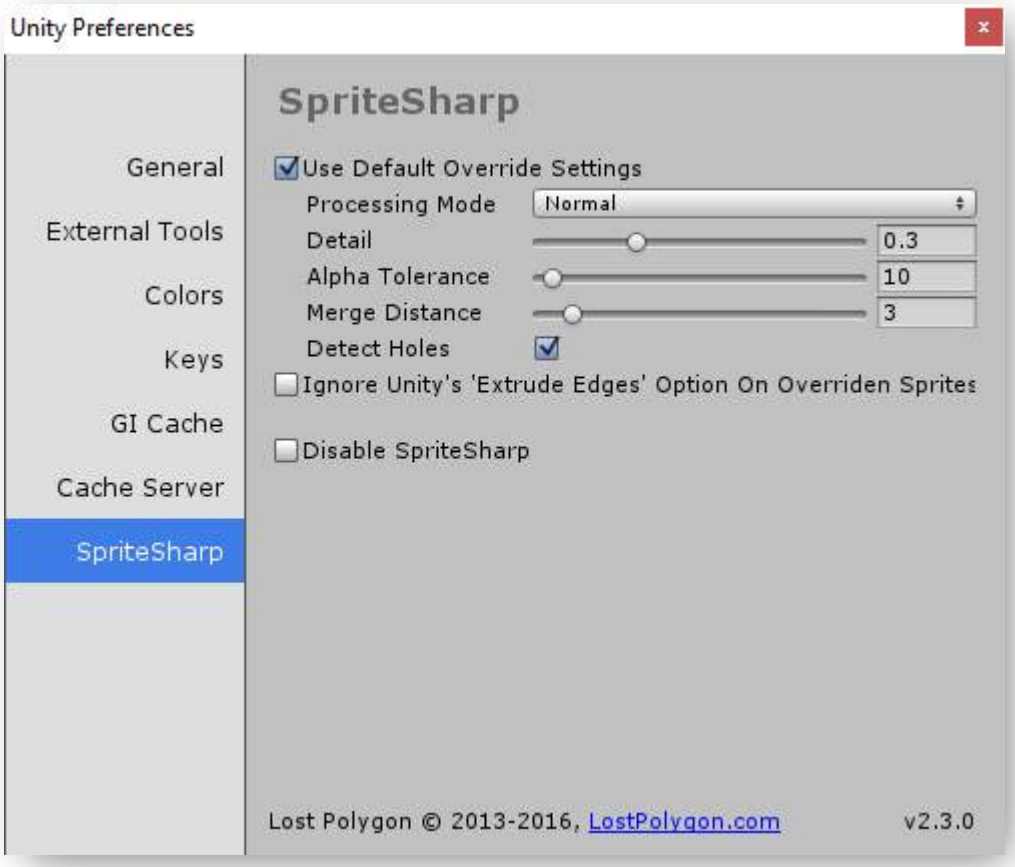
SCALE AROUND CENTER

Controls the amount of pixels the sprite mesh will be scaled in around sprite center.

Per-project Default Settings

SpriteSharp allows setting default sprite mesh settings to be used for all newly-added sprites and sprites without *Override* enabled. To edit these settings, go to

Edit → Preferences… → SpriteSharp

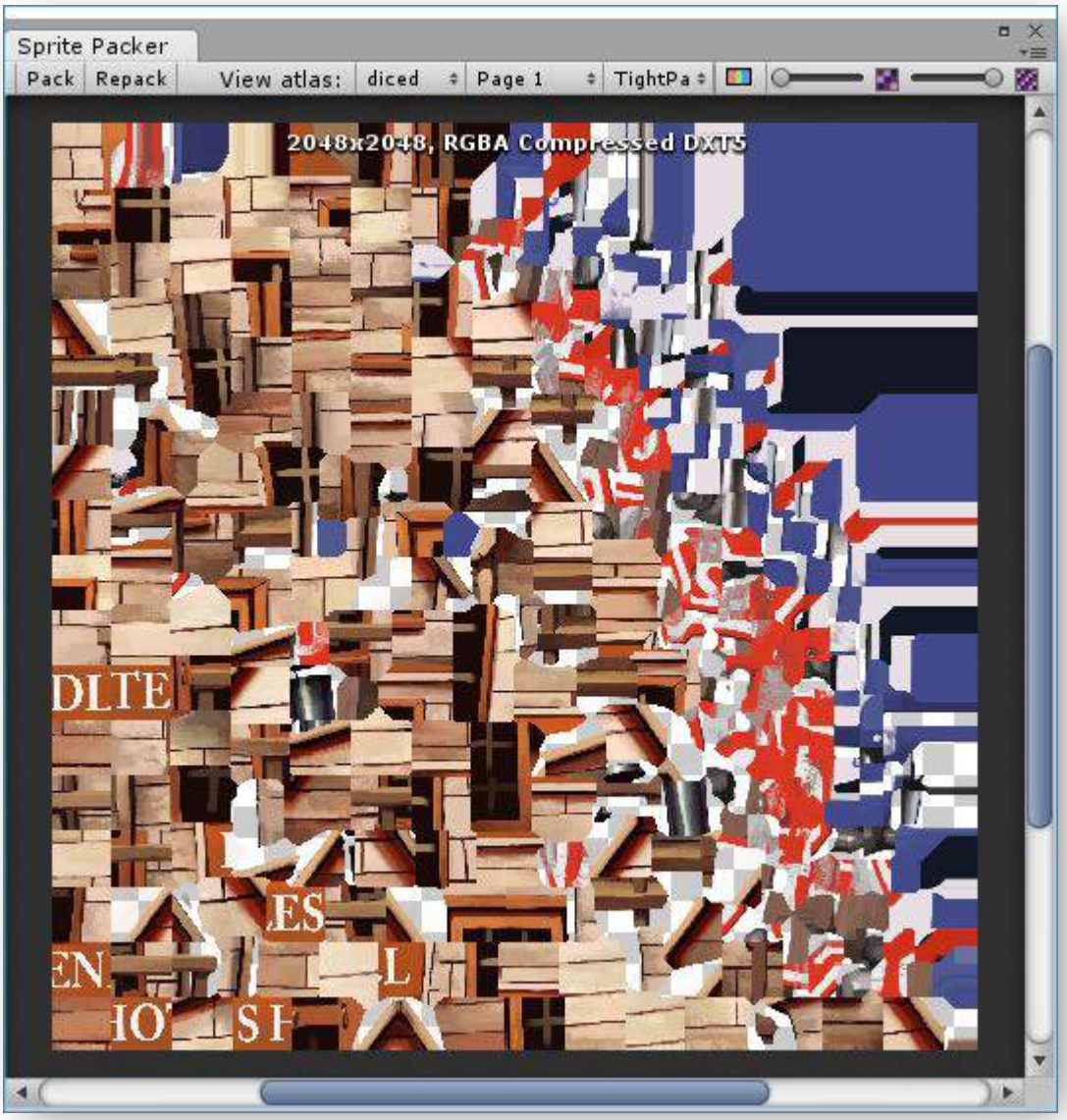


Sprite Dicing

Sprite Dicing is a feature that allows increasing the density of big sprites and images packed in atlases in exchange for increasing the polygon count. The idea is to split the texture into a grid of relatively small rectangles, which can be packed much more efficiently, especially when the original texture has complex shape. This allows packing multiple sprites with complex shapes in an atlas without increasing the atlas size.

As an example, let’s use this technique to pack these sprites:

And here is the packed atlas that contains all three sprites:

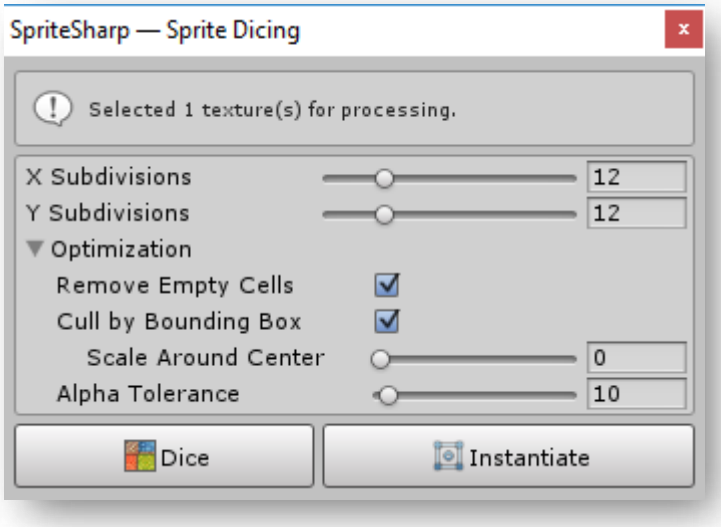


We've packed all that in a single 2048x2048 image, nice!

To open the Sprite Dicing window, use the menu item:

Tools → Lost Polygon → SpriteSharp → Sprite Dicing

Let's take a look at the interface.



First, you have to select the diced textures in the Project window. Multi-select is supported. Textures must have a packing tag set to be processed.

The dicing options are the same as used in the *Rect Grid* processing method. "Dice" button will dice the texture into smaller sprites. Transparent sprites will be omitted, and fully opaque sprites will have "Opaque" added to their name.

Note: the dicing operation *will remove all* sprites that were previously in the texture, and then create the dice sprites.

"Instantiate" button will insert a reassembled object into the scene.

Version Control

Database containing sprite mesh settings and per-project default settings is located at

`Assets/SpriteSharp/Editor/Database/SpriteSharpDatabase.json`

This file *has* to be under control of your version control system in order to share sprite mesh settings with your team members.

Source Code

SpriteSharp comes in both compiled DLL and source code forms, but right after importing the package you'll get the DLL version. It is highly recommended to use the DLL version in order to keep your project less cluttered and faster to compile, and avoid import inconsistencies in some cases.

To get the source code, import the package

```
Assets/SpriteSharp/SpriteSharpSource
```

Important: After importing the source code package, delete this file:

```
Assets/SpriteSharp/Editor/LostPolygon.SpriteSharp.dll
```

Contact

For any questions about this plugin, feel free to contact me at:

Unity forums thread: http://forum.unity3d.com/threads/spritesharp-sprite-mesh-optimizer.327997
E-mail: contact@lostpolygon.com
Skype: serhii.yolkin