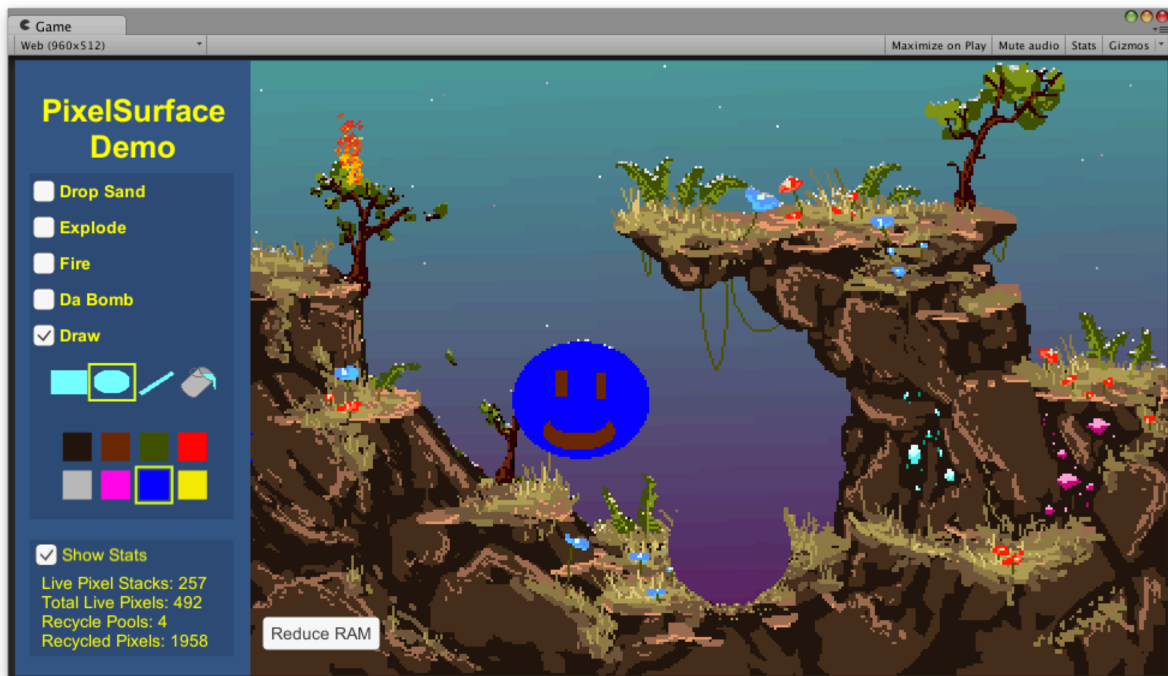
PixelSurface



a dynamic world of pixels for Unity

Oct 19, 2015

Joe Strout

joe@LuminaryApps.com

Overview

PixelSurface is a small class library for Unity that lets you manipulate 2D graphics on the level of individual pixels. You can efficiently draw lines, rectangles, and ellipses in full 24-bit color, as well as draw a texture (or just a portion of a texture).

But in addition to these standard pixel operations, PixelSurface also has built-in support for "live" (dynamic) pixels. These are pixels in motion — efficiently tracked, combined, and blitted for maximum performance. This makes it easy to dig tunnels, make destructible terrain, set things on fire, pour sand, and much more.

Namespace, Source Files, & Classes

Everything you need is defined in two files, PixelSurface.cs and LivePixel.cs. These contain the two primary classes, PixelSurface and LivePixel, both within a "PixSurf" namespace. (So, you may want to add "using PixSurf;" to the top of any file where you access these classes.)

PixelSurface is the main workhorse of the system. It is a MonoBehaviour which creates the pixel tiles, manages all the live pixels, and provides public methods for all the different kinds of drawing you might need to do.

LivePixel is a base class for any live (dynamic) pixels you define. It is *not* a MonoBehaviour, though in some ways it acts like one: you can override its Start and Update methods to execute custom code at the appropriate times. But LivePixel objects are designed to be considerably more lightweight than regular Unity objects, since a typical project may have many thousands of live pixels at once.

PixelSurface also defines one contained struct, PixelSurface.Stats. This is a simple container for several statistics (live pixel count, etc.) that you may want for debugging or performance-tweaking purposes.

Included Demo

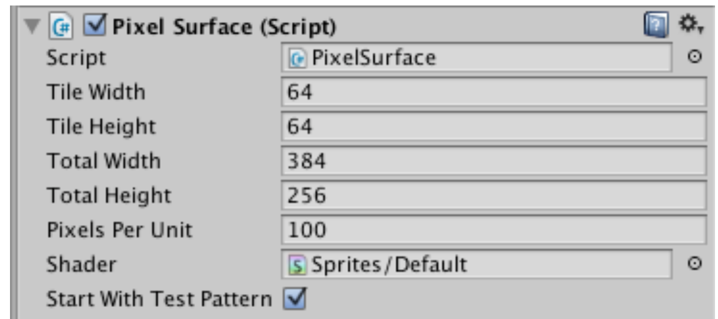
The PixelSurface package includes a test scene that illustrates use of PixelSurface and LivePixel. It is divided into a number of small scripts that do particular functions: DrawStuff, DropBomb, DropSand, MakeFire, and so on. It also includes several subclasses of LivePixel demonstrating different pixel behaviors (fire, snow, physics-like particle, etc.). Finally, the PixSurfDemo script brings it all together, automatically enabling or disabling the other demo components according as selected by the GUI.

You don't need to include any of those demo scripts in your own projects, though of course you are welcome to use or adapt them as you wish.

Getting Started

To create a new PixelSurface in your scene is simple:

1. Create an empty GameObject, positioned where you want the lower-left corner of the surface to appear in the world.
2. Add a PixelSurface component.
3. Adjust the tile PixelSurface properties in the inspector as desired.



The PixelSurface component inspector is shown at right. Let's examine each of these properties, starting at the bottom and working our way up.

Start With Test Pattern indicates whether you want the pixel surface to be initialized to a fractal pattern of triangles (checked), or to clear (unchecked). The test pattern is often useful because it can be hard to position a clear (and therefore invisible) pixel surface.

Shader lets you select the shader that should be used for drawing the surface. If this is null, then Sprites/Default (the standard sprite shader) will be used.

Pixels Per Unit defines how pixels are mapped to game world units. The default value of 100 means that every 100 pixels is 1 unit across.

Total Width and **Total Height** define the size of the pixel surface, in pixels.

To explain **Tile Width** and **Tile Height**, we must first explain that for performance reasons, a pixel surface is normally divided into multiple smaller tiles. This limits the amount of texture data that must be updated at once when something changes, and for parts of the pixel surface that happen to be all the same color, often lets us avoid using a texture at all. However, we pay a price in draw calls: each unique tile is its own material, and therefore a separate draw call. By adjusting these parameters, you can control the balance between efficient texture updating and draw calls. If you don't want tiles at all, simply set the tile size equal to the total size.

Once you've set the properties as desired, simply run your scene, and you have a working pixel surface! The **Start With Test Pattern** option is recommended at first until you have some code ready to draw something else.

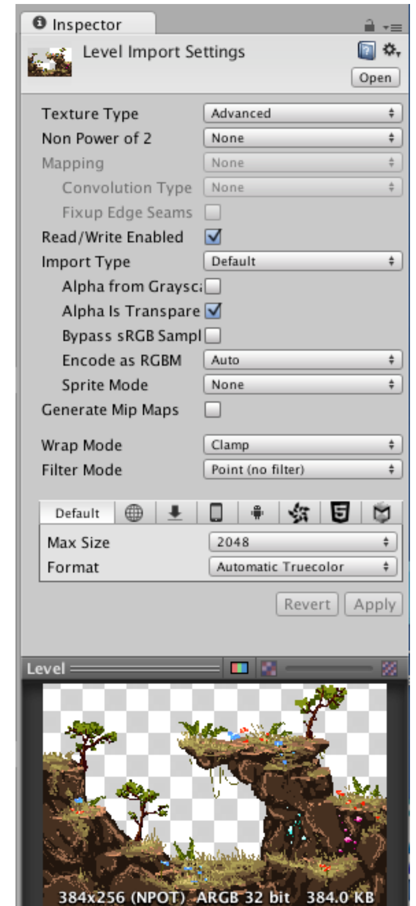
Drawing to a PixelSurface

This section provides an overview of the drawing methods you can use on a PixelSurface. For full details, see the Reference section.

The **Reset** method clears all pixels and data, returning the PixelSurface to pristine condition; it should also be called after changing the tile or total size. The **Clear** method

is similar, but does not rebuild the tiles or clear all the internal data; Clear can also take an optional parameter to clear the surface to a specified color.

Shapes can be drawn with **FillRect** and **FillEllipse**. 1-pixel-thick lines are drawn with **DrawLine**. You can also draw a texture, or any portion thereof, into any part of the surface with **DrawTexture**; note that if size of the source and destination rects are not the same, then the texture will simply be scaled as it is drawn. Also, note that any texture drawn has to be have "Read/Write Enabled" in the import settings (see example at right).



Individual pixels can be set with **SetPixel**, and read with **GetPixel**. Note that **GetPixel** has an "includeLive" parameter that controls whether you want to include live pixels when reading the color. By default, live pixels are ignored, and what is returned is the color of the static pixel at the given location. But if you pass true for includeLive, and there are any live pixels currently at that pixel location, then you will get the color of the topmost live pixel instead. Note that **SetPixel** has no such option; it always sets the color of the static pixel, even if that is currently under one or more live pixels.

FloodFill, as noted before, does a paint-bucket fill starting at a specified point.

Working with Live Pixels

A "live pixel" is an instance of LivePixel (or some subclass you define). Every live pixel is attached to a PixelSurface at some specific pixel position, and has a color. Multiple live pixels can occupy the same position; the set of live pixels at one position is called a "stack," and only the topmost (most recently created or moved) pixel in each stack is actually drawn.

You create a live pixel by calling **CreateLivePixel** on the pixel surface. This is a generic method that takes the specific type (which must be LivePixel or some subclass) that you want to create. Note that this method may actually reuse a previously killed live pixel from a recycling pool rather than instantiating a new one. In either case, the Start method is called; this is a virtual method, which you can override in your subclass to perform initialization.

Then, on every frame, the Update method is called for every live pixel on every active PixelSurface. The Update method is your chance to change the position or color of the live pixel; any such changes will be applied at the end of the frame. The Update method is called only once per frame, no matter how you move the pixel around.

When you are done with a live pixel, you should tell it to die. There are two ways to do this. The standard **Die** method on LivePixel first sets the static color of the PixelSurface at that position to the live pixel's color (unless the live pixel's color is clear), and then recycles the live pixel. The **DieClear** method recycles the pixel without applying its color to the surface.

If you want to quickly clear all the live pixels at a given pixel location, the **ClearLivePixels** method on PixelSurface will do the job.

Other Methods

PixelSurface contains several methods for converting between coordinate systems. **PixelPosAtWorldPos** returns the pixel position at a given world position, and **WorldPosAtPixelPos** converts the other way. These are very handy when combining a pixel surface with sprites (since sprites use world positions).

There is also **PixelPosAtScreenPos** to convert from a screen position (for example, what you get from Input.mousePosition) to a pixel position. Note that this function uses a ray-cast onto the pixel surface for maximum accuracy, but means that it can't return an answer at all if the given screen position is out of bounds (but the function has a return value to let you know when this is the case). Given a pixel position, **InBounds** provides a quick way to see if it's within the range of pixel positions for the surface.

The **GetStats** method returns a data structure providing various statistics about the pixel surface: the number of live pixel "stacks" (i.e. pixel locations that have one or more live pixels), and the total number of live pixels; as well as information about the recycling pools. Note that gathering these statistics does take some time, so you should only call **GetStats** when you really need it, and probably not in production code.

Finally, the **ReduceRAM** method is there to reduce memory usage when RAM is getting tight (for example, if you have received a low-memory warning on a mobile device). It drains the recycle pools, and may release other cached or temporary objects. There is generally a performance cost for doing this, so it should only be called when you really need the memory back. (Note that when a PixelSurface is destroyed, all that memory is freed anyway.)

Reference

void **Clear** ()

Clear the whole surface (to Color.clear). Also clears all live pixels.

---

void **Clear** (Color *color*)

Clear the entire surface to the given color. Also clears all live pixels.

Parameters

color	Color to fill the surface with.

---

void **ClearLivePixels** (int *x*, int *y*)

Clear all live pixels at the given location.

Parameters

x	The x coordinate.
y	The y coordinate.

T **CreateLivePixel**< T > (int *x*, int *y*, Color *color* = default(Color))

Create (or recycle) a LivePixel at the given position. The new pixel is attached to this pixel surface at the given position, and its Start method is called so it can initialize itself.

Returns

The newly created (or recycled) live pixel.

Parameters

x	The x coordinate.
y	The y coordinate.
color	Optional color to set.

Template Parameters

T	Specific type of LivePixel to create.

---

void **DrawLine** (Vector2 *p1*, Vector2 *p2*, Color *color*)

Draw a 1-pixel-thick line between the given points in the surface.

Parameters

p1	Pixel coordinates of one end of the line.
p2	Pixel coordinates of the other end of the line.
color	Color to draw.

---

void **DrawLine** (int *x1*, int *y1*, int *x2*, int *y2*, Color *color*)

Draw a 1-pixel-thick line between the given points in the surface.

Parameters

x1	The first x value.
y1	The first y value.
x2	The second x value.
y2	The second y value.
color	Color to draw.

---

void **DrawTexture** (Texture2D *src*, Rect *destRect*, Rect *srcRect*)

Draw a texture into the surface. You can draw any rectangular portion of the source texture into any rectangular area of the PixelSurface.

Parameters

src	Source texture to draw.
destRect	Where to draw the texture in this surface.
srcRect	What part of the source texture to draw.

---

void **DrawTexture** (Texture2D *src*, Rect *destRect*)

Draw a texture into the surface. This version draws the entire source texture into any rectangular area of the PixelSurface.

Parameters

src	Source texture to draw.
destRect	Where to draw the texture in this surface.

---

void **DrawTexture** (Texture2D src)

Draw a texture into the surface. This version fills the entire [PixelSurface] with the entire given texture.

Parameters

src	Source texture to draw.

---

void **FillEllipse** (Rect *rect*, Color *color*)

Fill an elliptical region with a color. Note that the coordinates in the bounds rectangle are truncated to the next lower integer.

Parameters

rect	Bounds rect within which to inscribe an axis-oriented ellipse.
color	Fill color.

---

void **FillRect** (Rect *rect*, Color *color*)

Fill a rectangular region with a color. Note that the coordinates in the rectangle are truncated to the next lower integer.

Parameters

rect	Rect to fill.
color	Color to fill.

Color **GetPixel** (int *x*, int *y*, bool *includeLive* = `false`)

Get the color of the specified pixel. If out of bounds, returns Color.clear. Normally this method ignores any live pixels at the specified position, but if includeLive=true, then you will instead get the color of the topmost live pixel.

Returns
Pixel color at the given x,y.
Parameters

x	The x coordinate.
y	The y coordinate.
includeLive	If set to `true` include live pixels.

Color **GetPixel** (Vector2 *pixelPos*, bool *includeLive* = `false`)

Get the color of the specified pixel. If out of bounds, returns Color.clear. Normally this method ignores any live pixels at the specified position, but if includeLive=true, then you will instead get the color of the topmost live pixel.

Returns
Pixel color at the given pixel position (rounded to nearest integers).
Parameters

pixelPos	Pixel position.
includeLive	If set to `true` include live.

Stats **GetStats** ()

Get the current statistics for this pixel surface, for debugging or analysis (or just plain curiosity).

Returns
The stats.

---

bool **InBounds** (Vector2 *pixelPos*)

Return whether the given pixel position is within bounds of this pixel surface.

Returns
`true`, if within bounds, `false` otherwise.
Parameters

pixelPos	Pixel position of interest.

---

bool **PixelPosAtScreenPos** (Vector3 *screenPos*, out Vector2 *pixelPos*, Camera *camera* = `null`)

Get the pixel position at a given screen position (relative to the given camera, or Camera.main if none is specified). Handy for finding the pixel position clicked, for example.

Returns
`true`, if screen position is in bounds; `false` otherwise.
Parameters

screenPos	Screen position of interest.
pixelPos	Receives the corresponding pixel position.
camera	Optional camera of interest (uses Camera.main by default).

Vector2 PixelPosAtWorldPos (Vector3 worldPos)

Get the pixel position at a given world position (ignoring the local Z direction).

Returns
Pixel position corresponding to the given world position.
Parameters
worldPos World position of interest.

---

void **ReduceRAM** ()

Reduce our current RAM usage as much as we can by freeing recycled objects, etc.

---

void **Reset** ()

Reset this pixel surface, clearing all pixel data.

---

void **SetPixel** (int *x*, int *y*, Color *color*)

Set the static pixel color at the given position. Note that if there are any live pixels at this position, those are ignored in this operation; you're only setting the static color here.

Parameters
x The x coordinate.
y The y coordinate.
color Color to set.

Vector3 **WorldPosAtPixelPos** (Vector2 *pixelPos*)

Get the position in the world of a given pixel position.

Returns
World coordinates of given pixel position.
Parameters
pixelPos Pixel position of interest.