Sand Game Template Manual

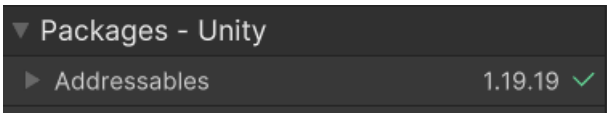
Table of contents

Requirements

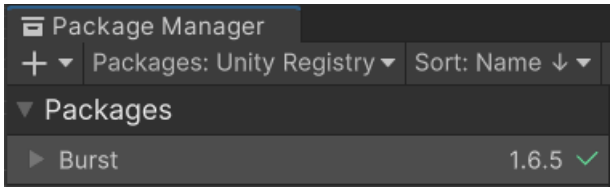
Unity 2021.3 or higher is required. If you can, please upgrade to the highest LTS version of Unity. The newer the version the less „glitches“ it usually has.

Required Packages (all freely available):

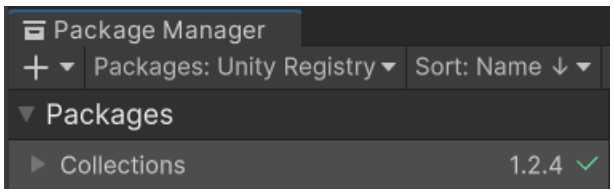
- Addressables (official Unity package)



- BURST (official Unity compiler)

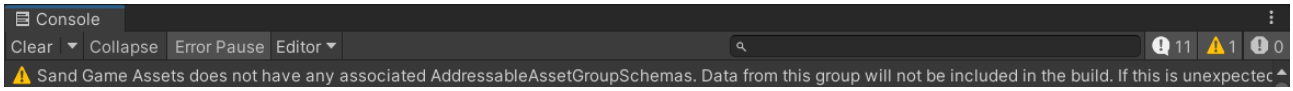


- Collections (official Unity Collections for BURST)

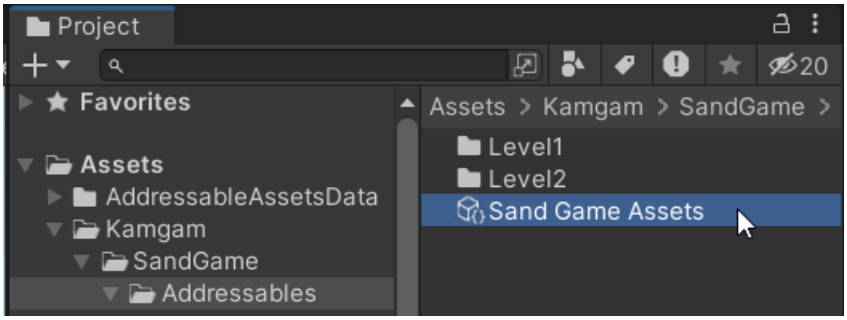


Installation

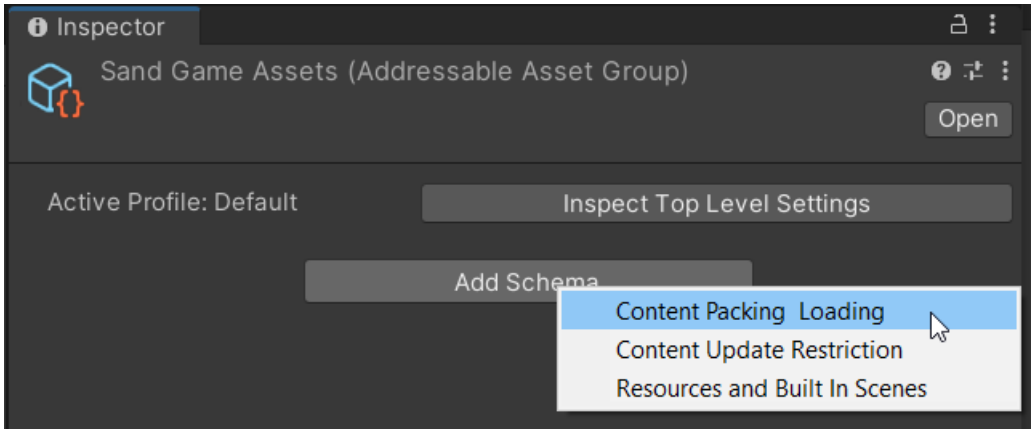
Depending on your Unity version you may have to add a schema to the „Sand Game Assets“ group asset.

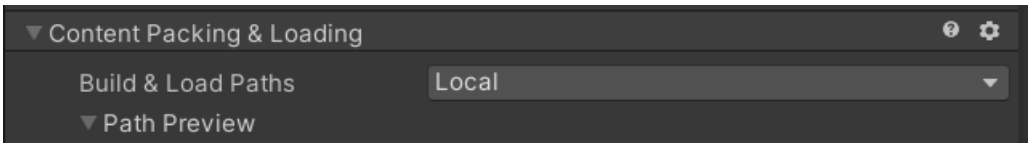


To do this please find the asset and add a schema that fits your needs.

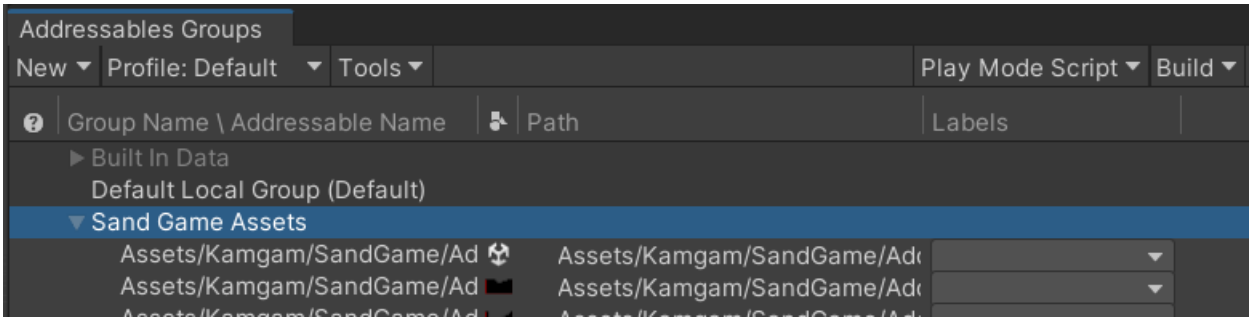


Usually adding a „Content Packing Loading“ is enough to make it work.





Please also check that the asset group and the assets within are listed in the addressable groups (Window > Asset Management > Addressables > Groups):



Intro (read this first)

Since this asset uses BURST and Jobs it is structured quite differently from your normal Unity project. This manual assumes you have familiarized yourself with how BURST and Jobs work in Unity. The Official DOTS Github page is a good place to start learning.

Please notice that this is NOT a fully featured game like Noita. It's a template for you to build your game upon. There is still lots of stuff you will have to implement yourself (rigibodies, particles, lighting, world streaming, savegame).

This projects uses some concepts presented by others who have done sand games before.
I recomment you start start by visiting some (or all) of these:

How To Code a Falling Sand Simulation:
https://www.youtube.com/watch?v=5Ka3tbbT-9E

Exploring the Tech and Design of Noita (GDC):
https://www.youtube.com/watch?v=prXuyMCgbTc&t=322s

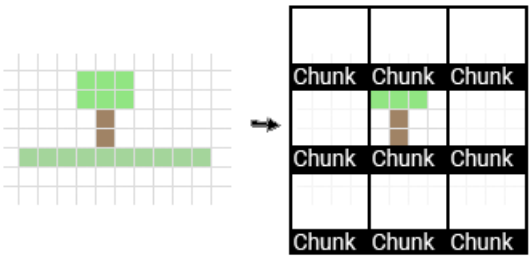
How To Code a Falling Sand Simulation:
https://www.youtube.com/watch?v=5Ka3tbbT-9E

Recreating Noita's Sand Simulation:
https://www.youtube.com/watch?v=VLZjd_Y1gJ8

An Exploration of Cellular Automata by Seppe "Macuyiko" vanden Broucke:
https://blog.macuyiko.com/post/2020/an-exploration-of-cellular-automata-and-graph-based-game-systems-part-4.html

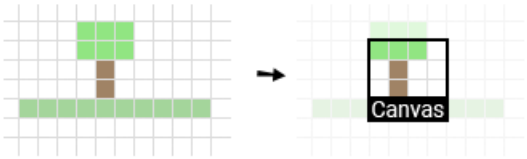
Pixel World Overview

Theoretically the pixel world is endless. To deal with this we divide it into small rectangles called „chunks". The content of each chunk is loaded from a small image (called „level part").



However, in the game we are usually only interested in the small part that is shown to the player. That area is often called the „viewport". That viewport is rendered into what we call a „canvas" (basically a render texture).

It's no coincidence that the viewport is roughly the same size as a level chunk (makes loading easier and more efficient).

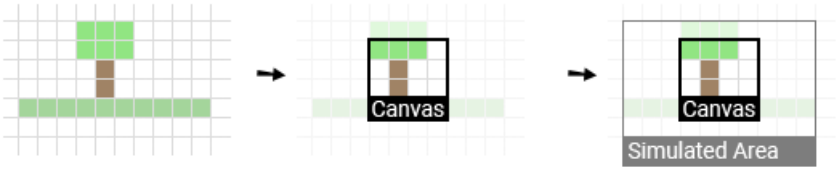


Though, we have a problem.

If we only simulate the pixels inside the viewport then things like water will stop flowing as soon as we do not look at them.

„Why not simulate the whole world all the time?" you might ask. Well, that's not possible because it would require too much memory and processing power. That's why we are compromising.

We do simulate just a bit more than we show:



In code the „Simulated Area" is a NativeArray and is called „_simulationBuffer".

If you move very fast you may encounter some pixels that have just now started to simulate but in most cases this should be fine.

This simulated area is what we will call the „simulation buffer". It is a list of all the pixels that are being simulated right now (during this frame).

So the steps from static level images to the final rendered result are:

1) Copy pixels from level parts (png images) to chunks (memory) – OR – use the existing chunk data from the previous frame.

2) Find the viewport area and copy the pixels inside (and those surrounding it) into the simulation buffer.

3) Simulate all pixels in the buffer.

4) Copy the buffer results back to the chunks so we can use them as a starting point next frame.

5) Draw the pixels inside the viewport on the canvas (copy in to a render texture).

Learning from other games

At first glance the whole world could be represented in a single pixel grid. Yet if we look closely at games like Noita we can see that there are multiple overlapping systems at play.

It is desirable that some objects can move smoothly outside the pixel grid.

To create the illusion of a coherent world things like particles may be taken out of the pixel grid.

Example: In Noita particles are simulated without interacting with the pixel world. Once they hit something they are destroyed and replaced by a „real" pixel in the world that interacts with its surroundings. This generates the illustion that the particle is part of the world, yet while flying it can be smoothly animated and rotated.

In Noita this scheme is also be applied to many other elements like items, characters and some enemies. It is used for anything that is „dynamic".

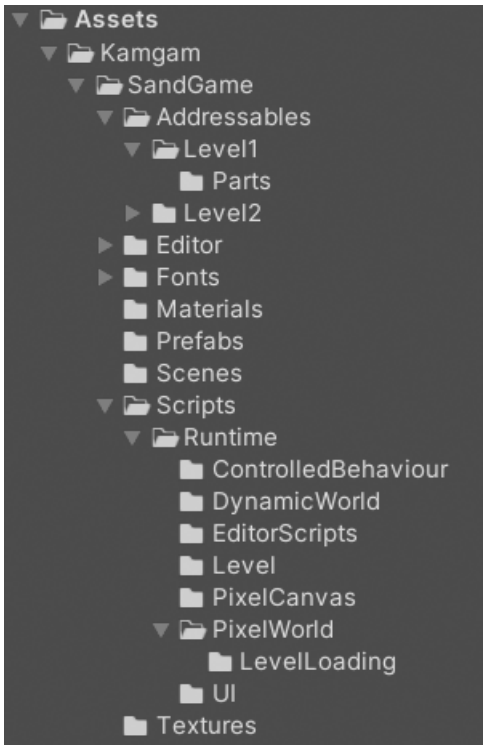
Naming conventions:

DYNAMIC: Anything that is not exclusively controlled by the Pixel Simulation is called „dynamic". These are things like the main character, enemies, particles, items and physics driven elements.

STATIC: Anything that is controlled only by the Pixel Simulation is called „static". While they are anything but static these things do not interact directly with anything outside the simulation.

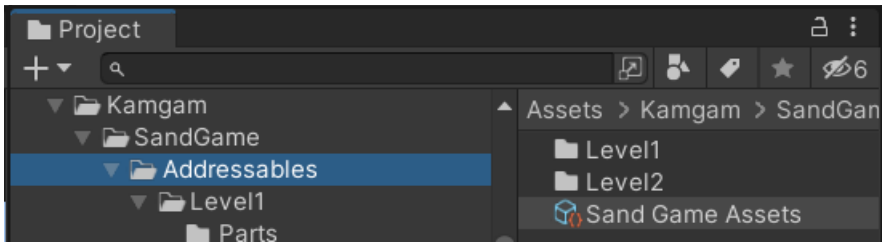
File Structure

When you first open the project you will find a structure like this:

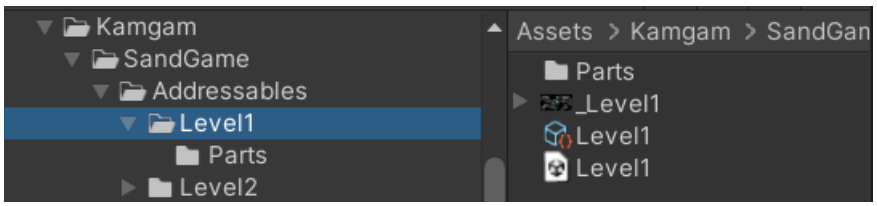


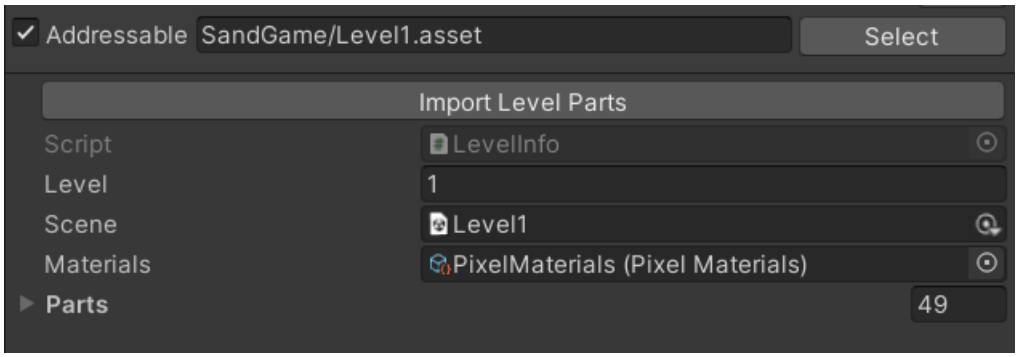
Assets

The assets in the template are organized as Addressables. By default these are stored in the „Sand Game Assets" group.



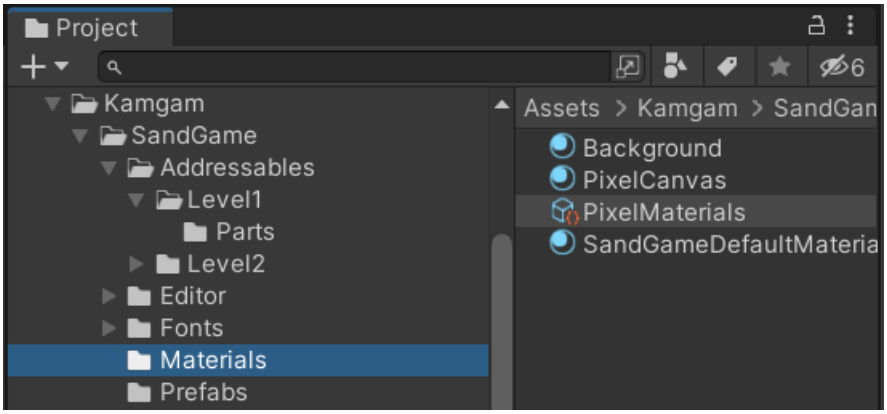
Each Level has a folder containing the level parts as images (pngs in the „Parts" subfolder), a Unity scene and a scriptable object for meta data.

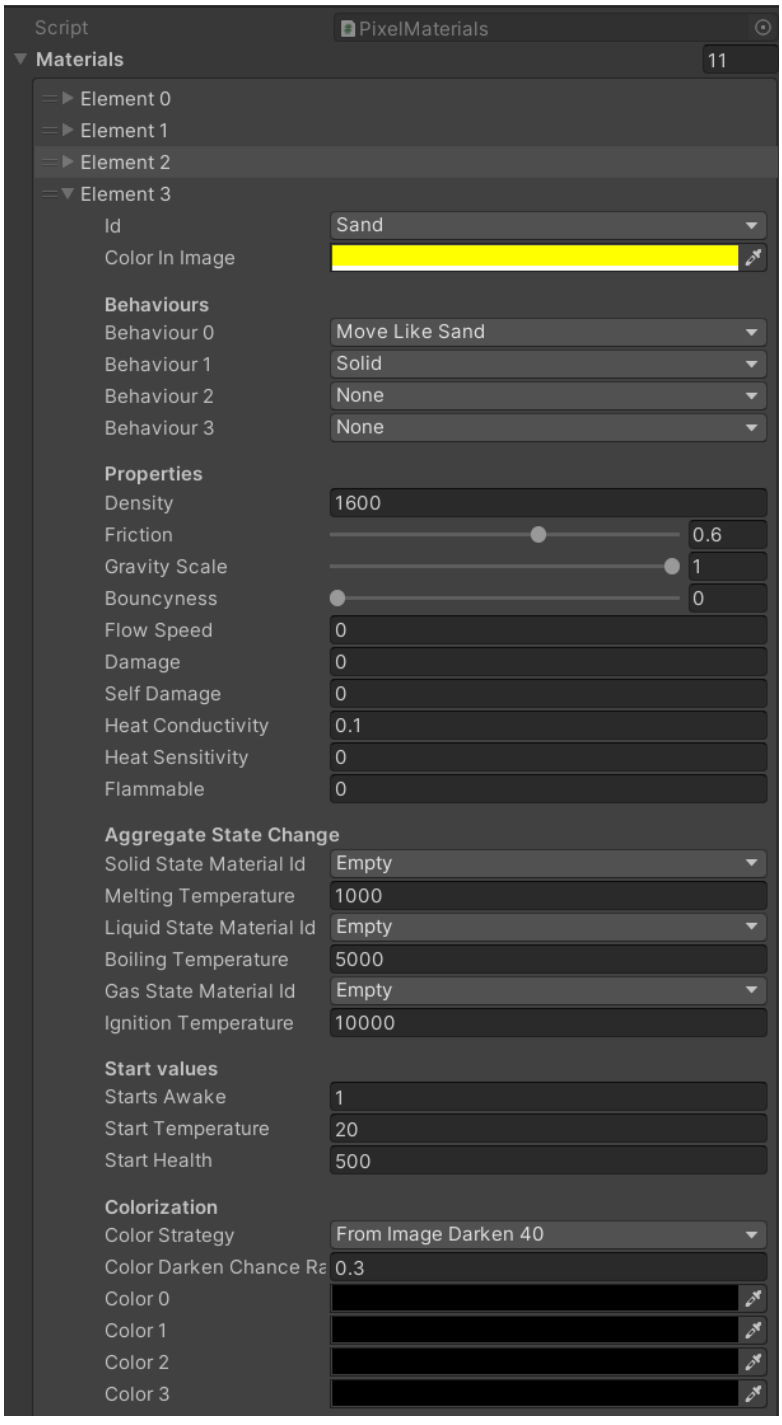




Materials

In the „Materials“ folder you will not only find the usual Unity materials but also the „PixelMaterials“ object which contains infos on all the materials supported in the pixel world.





Code Overview

Overall the class hierarchy looks like this:

SandGame → SandWorld → PixelWorld → PixelCanvas

Frame Lifecycle

To make the pixels move they have to be simulated. This involves many steps. Some of them are multithreaded. Some can take multiple frames to complete (loading level images) while others need to complete every frame. Some parts are not yet implemented (TODO).

Here is when things are executed in relation to the default Unity frame lifecycle:

FixedUpdate:

Default Unity fixed update stuff (physics update) & TODO: apply forces from previous frame.

Update:

ControlledBehaviour.OnUpdate()

LateUpdate:

Wait for „loading pixel world chunks" from previous frames. (This is not blocking. It asks the jobs if they are done and if yes then it triggers some code – see „LevelInfo.JobAwaiter")

Start loading pixel world chunks from images (*multi-threaded*, may take multiple frames)

TODO: Copy dynamic scene objects to pixels (*BURST job on main-thread*).

Copy pixels from world chunks to simulation buffer (*multi-threaded via CompleteAll*)

Simulate pixels in buffer (*multi-threaded via CompleteAll*)

Copy simulation buffer back to pixel world chunks (*multi-threaded via CompleteAll*)

Copy pixels from world chunks to canvas texture (*multi-threaded via CompleteAll*)

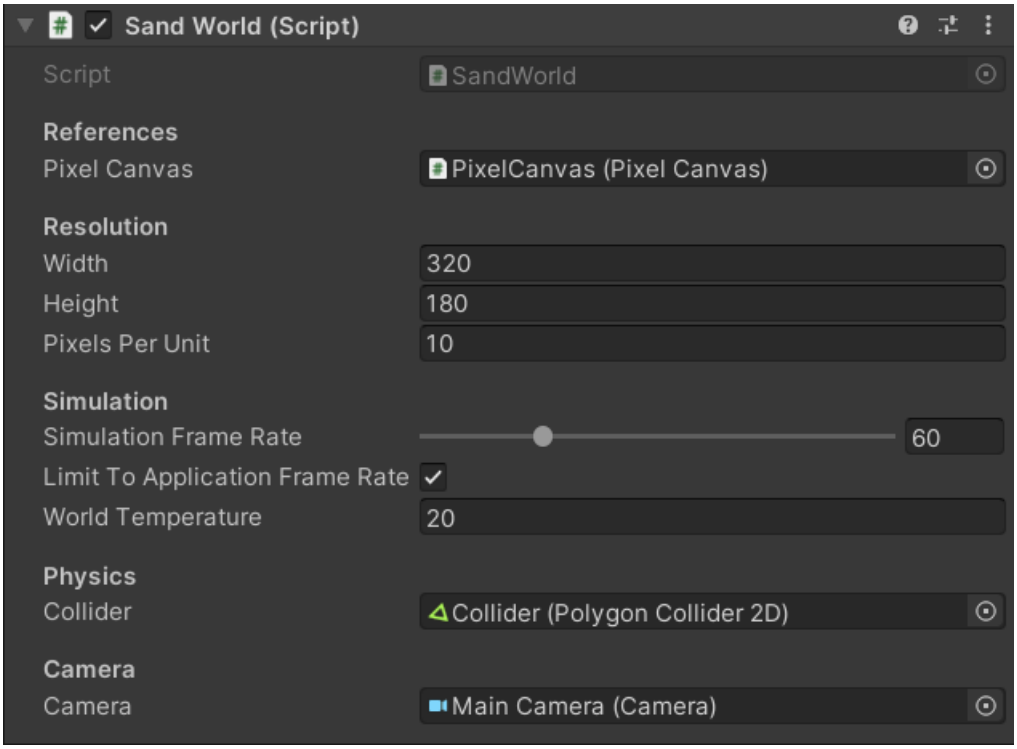
Generate PolygonCollider2D from the pixel world & TODO: queue physics forces.

NOTICE: If a chunk is being written to by the load from image thread then that chunk is skipped from processing until copying from the image is done (i.e. chunks are only simulated if they are fully loaded).

Sand World > Pixel World

Conceptually the pixel world is one giant 2D pixel grid that is upated every frame in LateUpdate(). Late update was chose so the pixel world can (in theory) react to any changes you have made during the regular update.

The origin position of the pixel world (x=0, y=0) matches the Unity scene origin (x=0, y=0, z=0). Each scene unit is diveded into a certain number of pixels („PixelsPerUnit").



The „SandWorld" is the root component of the pixel simulation. It passes many of its parameters on to the PixelWorld object.

Pixel Canvas: A reference to the pixel canvas object in the scene. The canvas basically is a render texture on a quad in the scene. The pixel simulation results are copied into that render texture and the quad displays it.

Width/Height: The world chunk resolution (how many pixels are approximately visible on one screen). Your level parts have to match this resolution!

Pixels Per Unit: How many pixels in the a chunk are 1 Unit in the scene (aka 1 meter).

Simulation Frame Rate: The maximum frame rate at which the simulation will run. It works similar to fixed physics updates. Meaning it may run 0, 1 or multiple times per frame.

If you increase the frame rate then then simulated pixels will be sped up in comparison to free falling pixels.

The simulation code is tweaked to give nice results at 60 fps. Do NOT change this unless you absolutely need to.

This is a tricky parameter. You should try to keep it in sync with your Application.targetFrameRate. If the frame rate of the application drops below this then the system may try to run multiple simulation steps per frame. If the frame rate is much higher then there may be frames in which no simulation step is run.

Limit To Application Frame Rate: If enabled then the simulations actual simulation frame rate is capped to the real application frame rate (i.e. the simulation is executed at most once per frame).

If the frame rate drops too much then this may lead to diverging speeds between simulated and not simulated content (simulated pixels -VS- Unity scene objects).

World Temperature: If you add materials like snow then this is a relevant factor. It defines the temperature that each pixel will tend towards (i.e. snow will melt).

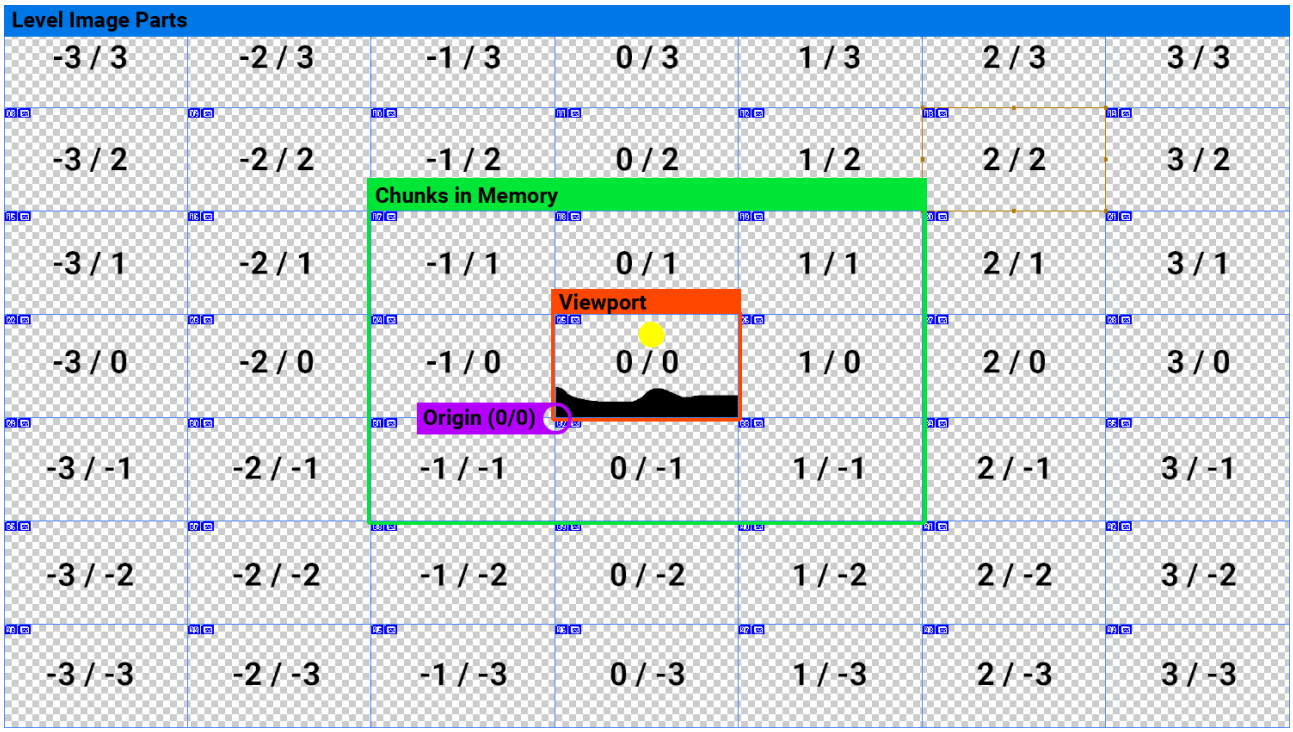
Collider: The PolygonCollider2D that will be updated based on the solid pixels in the world. NOTICE: The collider has some max size and thus you may have to manage object outside that size to prevent them from falling after the collider has vanished.

Camera: The camera that is used for rendering (has to use orthogonal projection).

Pixel World Chunk

To make the pixel world easier to handle it is divided into chunks. Each chunk is „Height“ pixels heigh and „Width“ pixels wide.

Initially only 9 chunks are loaded. The other chunks are added on demand based on which of them will become visible next.

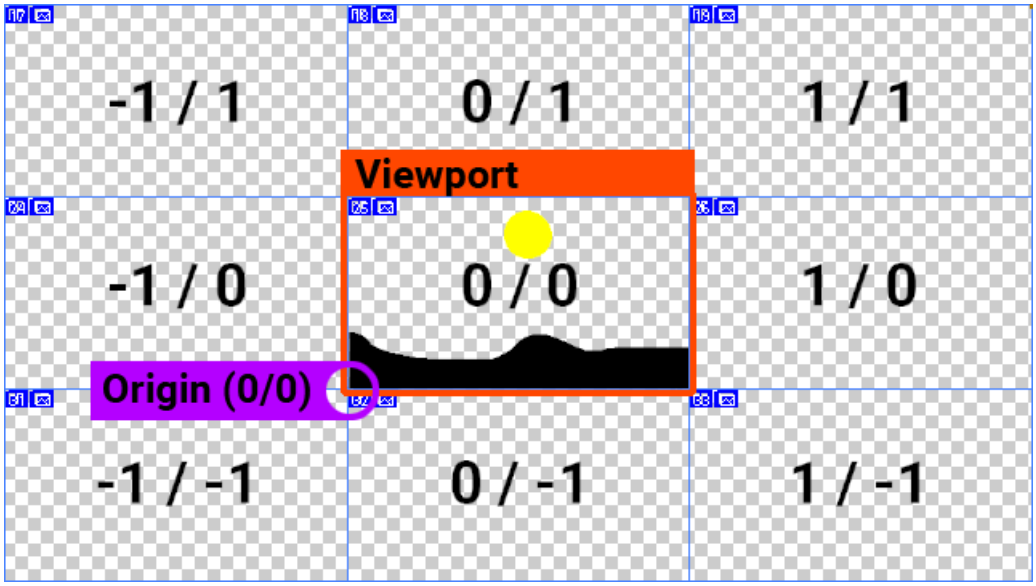


By default each chunk is 320x180 pixels and is based on one level part image. Level part images ave to match the chunk resolution exactly.

NOTICE: Level chunks are never unloaded (unless the whole level is unloaded). This means the bigger your level the more memory it will consume.

Pixel Viewport

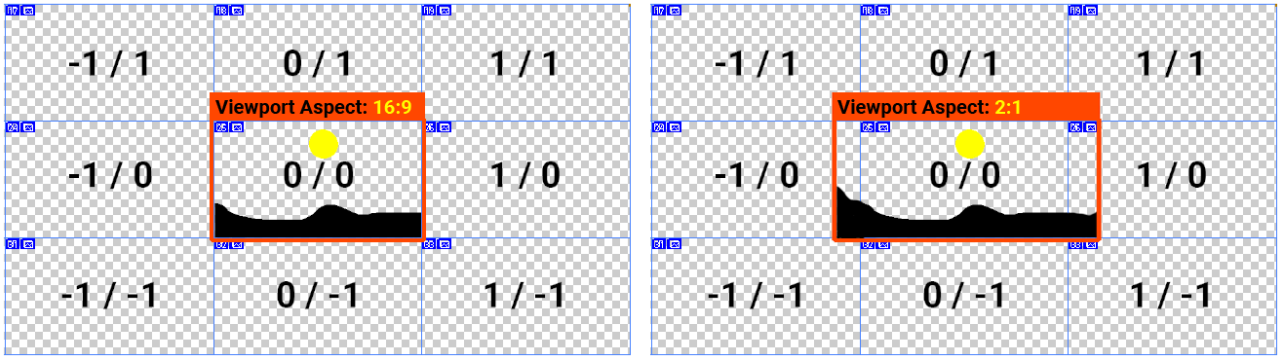
The viewport defines what part of the pixel world is shown on the canvas. In code the viewport is nothing more than a position (x/y) and a size (width/height). The position of the viewport is always measured from the bottom left corner (origin).



Internally the part of the pixel world that is selected by the viewport can only ever change by whole pixels. However, the viewport position can move smoothly by sub-pixel distances (more details on this are in the „Pixel Canvas" section below).

HINT: You can set the viewport position as float values (sub-pixels) to allow smooth animation of the viewport. The pixel canvas takes these float positions into account. - Usually you will move your 2D Camera in the scene and then ask the pixel world to align the viewport with the camera. This alignment is done with FLOAT precision (important for smooth animations).

The height of the viewport matches the pixel world chunk height (default: 180 px). The viewport width changes depending on the camera aspect ratio. Doing this ensures that it always covers the whole screen and thus avoids letterboxing.



The viewport will always grow or shrink from the center.

Pixel Canvas

The canvas is where the pixels inside the viewport area are rendered to (copied into a render texture). The canvas' render target is a simple quad in the Unity scene. The Unity 2D camera renders this quad together with your normal 3D scene.

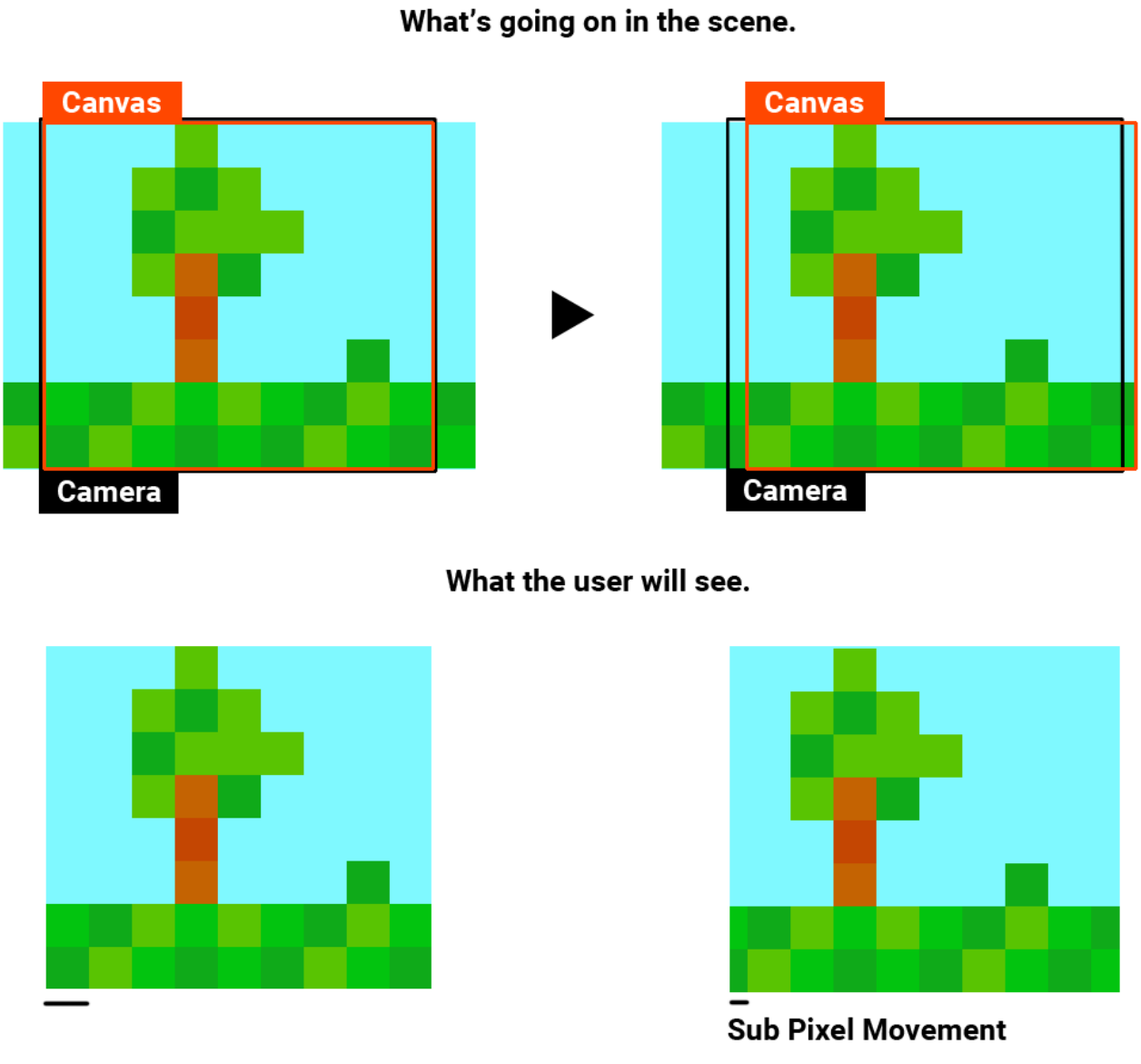
Appart from copying the pixel color from the pixel world the canvas also takes care of smoothing the camera movement.

Smooth sub-pixel camera movement

NOTICE: The part that is displayed in the canvas is always in whole pixels.

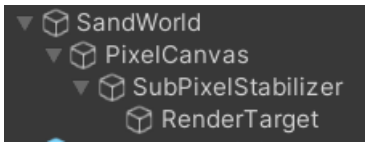
To fake smooth movement we shift the whole canvas relative to the 2D camera in the Unity scene.

This is done every frame to emulate smooth sub pixel movement. In truth only the canvas is moved to cover the fractional displacements of the viewport/camera.



You may have noticed that for this to work we need one extra pixel outside the canvas to compensate for the displacement. In the actual implementation the canvas is two pixels wider and higher than the viewport, giving use those extra pixels we need.

INFO: The actual movement of the canvas is applied via the „SubPixelStabilizer" transform.



Pixel Simulation (Cellular Automata)

These kind of simulations are called cellular automata (CA) in scientific literature. This simulation is at the core of the template. Every frame it updates each pixel. It is implemented using Jobs and BURST to allow high performance while manipulating millions of pixels.

To implement new game mechanics you have two options. The first one is to extend the existing pixel mechanics. Doing that will require you to use Jobs and BURST to keep performance high.

The second way is to add regular game objects and then have them interact with the pixel simulation. This one is easier but is limited to existing pixel mechanics.

Simulation Resolution

The simulation resolution defines how many pixels are shown, and thus simulated, in the world. This has a very big impact on the performance of the game since even only 320 * 180 will result in about 60.000 pixels to be simulated. Choose your resolution wisely.

NOTICE: This is NOT the actual render resolution. The render resolution is controlled by the default Unity mechanism. The simulation resolution height (180 pixels) is constant (output is streched to match the height of the screen). The simulation resolution width is cacluated based on the camera aspec ratio. So in truth you will rarely have the 320x180 resolution (unless you constrain the aspect ratio to 16:9). However, the chunk resolution (the images based on which the pixel world will be stitched together) will always have to match the given resolution.

Simulation Update Rule & Update Order

In order to explain how the simulation works we need to define some terms:

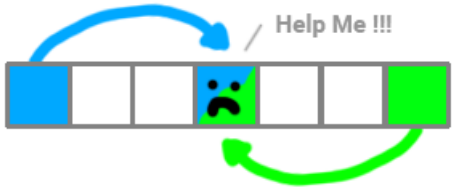
Update Rule: This is how each pixel changes per simulation step (frame). It is defined by the behaviours added to a PixelMaterial.

Update Order: This is the order in which the update rule is applied (which pixel gets changed first). The order has significant influence on the simultion result.

In our simulation we apply the update rule to the whole pixel grid in order and in place. This means each pixel will see the state as it is after the previous pixel has changed it.

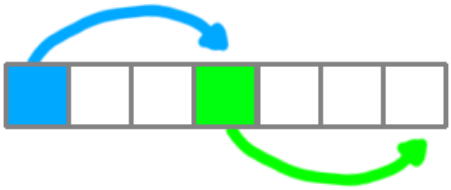
For each pixel in the pixel world we try to apply the update rule.

However, it may happen that two or more pixels will try to move to the same target location.

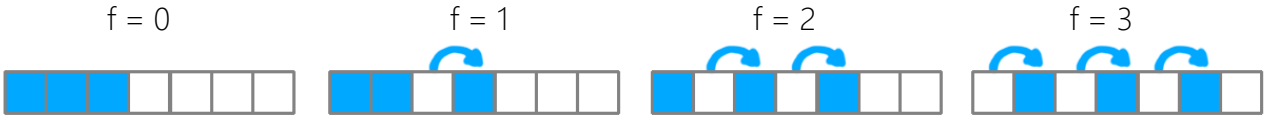


These cases are automatically resolved by the update order (either green or blue moves first).

It may take two simulation steps to resolve. For example (below) if blue wants to move first (update order left to right) then it will not move in the first step, though it will move in the second step once green has moved out of the way.



The update order has significant impact on the simulation. For example let's assume the update order is from LEFT to RIGHT. The images below show what will happen if we try to move the pixels to the right each frame (f).



As you can see the block will be spread out since only the pixels with empty spaces on the right can move. If the pixel on the far right stops moving then it will compress again.

However, this kind of streching behaviour can be reduced or eliminated in one direction.

If for example the pixels above were updated not from LEFT to RIGHT but from RIGHT to LEFT then they would all move without forming any gaps.

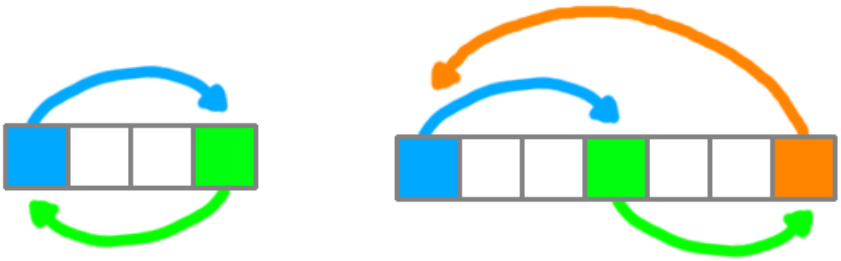
The thing is that this works for one direction on each axis (left-to-right or right-to-left). It also requires that in the direction of ordering (horizontal or vertical) there is only one thread -OR- the threads are ordered themselves. If random threads are used then the order would be messed up (gaps or vanishing pixels would be the result).

The update order chosen in most games is BOTTOM to TOP and LEFT to RIGHT to support the most common movement directions (falling from top to bottom due to gravity).

If you need to have groups of pixels that move together as a block in any direction without holes then you will have to handle them manually (avoid self-collision, check and resolve movement).

How do we handle loops?

Examples A and B form a loop.

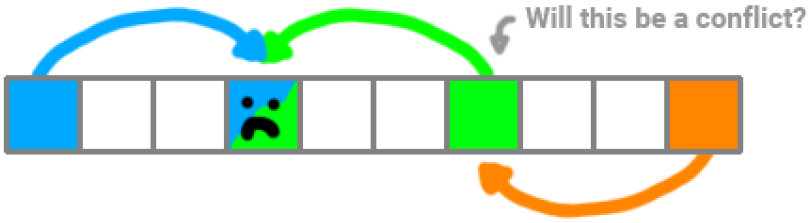


A) B)

If we start with the blue pixel then this would have to check if green can move, which then would have to check if blue can move, which would have to check if green can move, … . They form a loop! It would require significant processing per pixel (the longer the loop the more checks are needed).

To keep the implementation performant it was decided that these cases (A and B) consitute an acceptable flaw in the simulation (i.e. we don't perform any smart checks → nothing will move).

In some cases the movement might not be resolvable in one step. In the image below we see that whether or not the orange pixel can move to the green position depends on who wins the conflict between blue and green on the left.



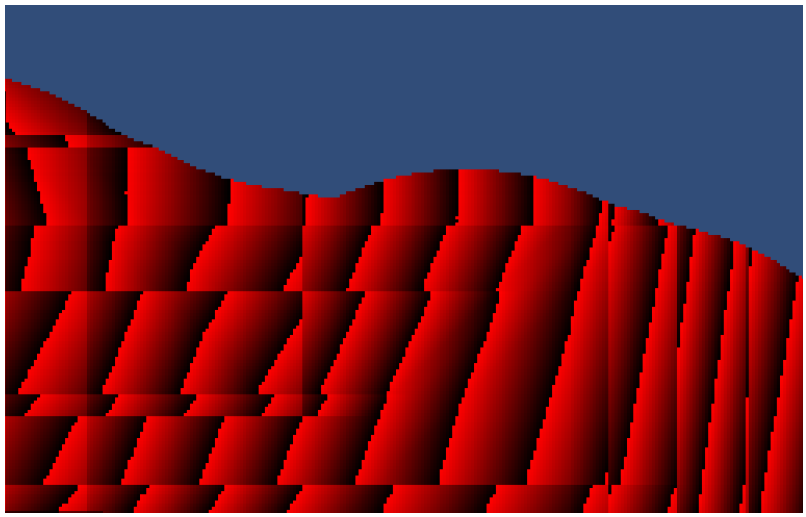
The difference between this and the loops from above is that this may resolve itself in the next simulation step: Orange will move in the next step if green has moved first in the previous step.

Global Simulation Synchronization (we don't have any)

Let's say you have an area in your level that has the ground color animating from black to red in a loop. You do this via the pixel simulation (important!).

At first everything looks fine:

But as soon as you move things start to shift and form blocks. Why is that?



The reason is that not all pixels in the world are updated all the time. Doing that would cost too much performance. Instead only the pixels inside or near the viewport are simulated.

Now, if one pixel is in the simulated area but the one next to it is not then only the pixel inside will be simulated and thus change the color. The pixel on the outside remains the same color. This causes the staggered effect you see in the image above. The pixels get more out of sync the more you have moved around. With every move of the viewport some pixel stop simulating while others start.

HINT: Getting a stable simulation behaviour over all distances can be achieved if you make your simulation dependent on a uniform factor like TIME. The simulation result should be like a function: f(t) = always same result if all ese is equal. That way you can sync the colors based on a value that is the same across the whole pixel world instead of the unpredictable simulation step count.

You should never rely on the simulation update loop for global synchronization. Doing that is a bit like relying on the FPS for animations. TIME or POSITION are good alternatives for global synchronization.

Pixel Movement & Time.deltaTime

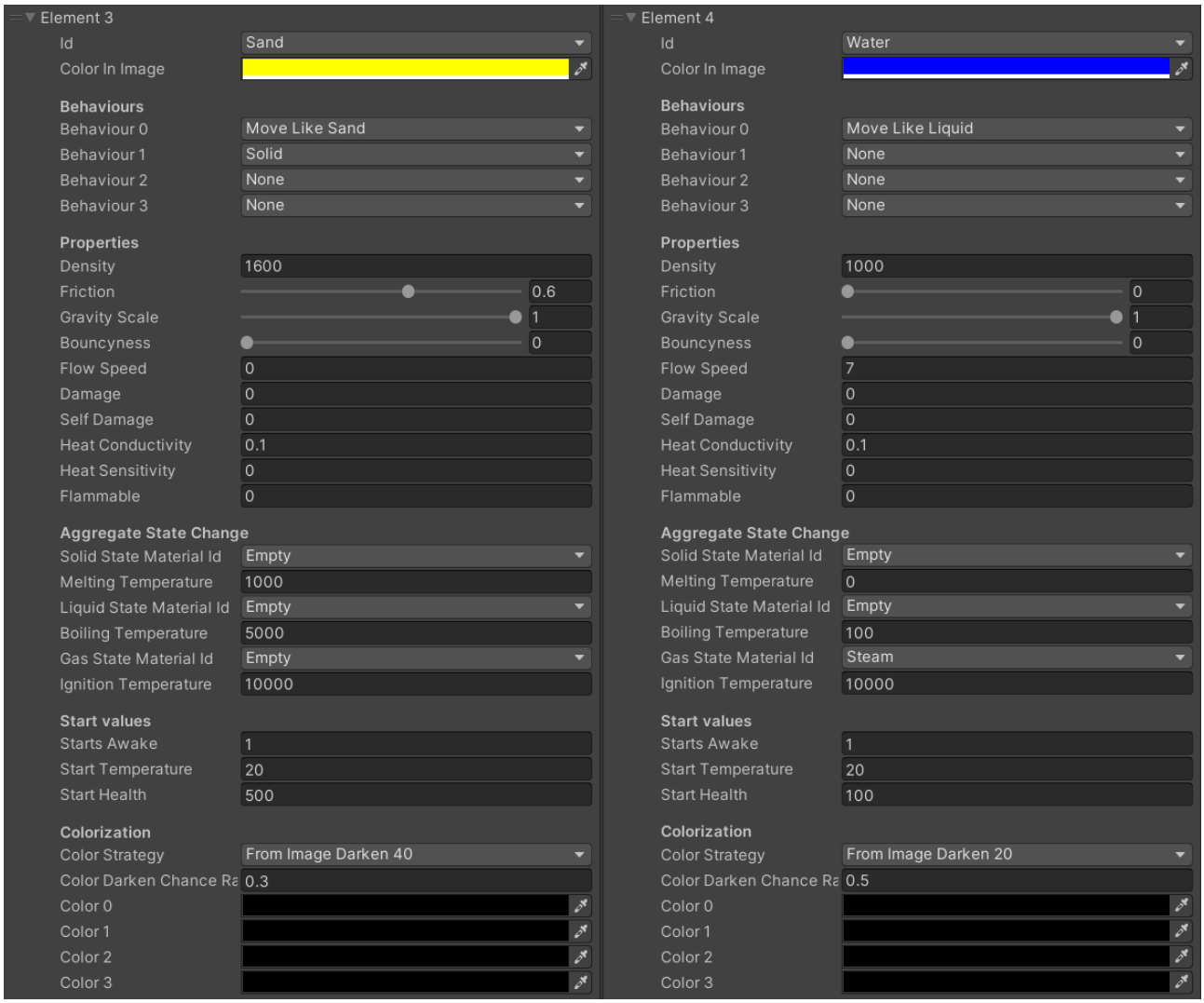
As you are probably aware movement is usually created frame independent by using Time.deltaTime as a multiplier for the distance travelled within a frame.

Since the movement of a pixel is constrained to a grid it can only move if the distance is bigger or equal to the distance between two pixels. Though even then the movement will never be 100% accurate in relation to time (except if by chance the distance travelled should be exactly 1 pixel).

To compensate for that each pixel has a deltaX and deltaY value stored as a FLOAT. These can be animated smoothly (with Time.deltaTime taken in to account). At the end of each frame these values are checked and if > 1 then the pixel will move and the delta is reduced. That way you can have pixels moving in less or more than increments of one.

Pixel Materials

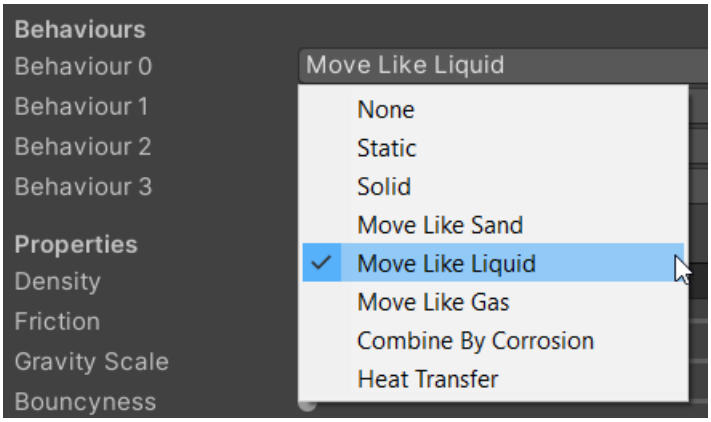
How a pixel interacts with other pixels is defined by the behaviours of its material and material properties. Each material has a ID (enum) which can be used to easily distinguis them from each other.



Pixel Behaviours

Behaviours define the logic of pixel to pixel interaction and movement. A pixel can have more than one behaviour at the same time (movement and corrosion for example).

Some behaviours required additional data which can be entered in the form of material parameters. Example: The behaviour „heat transfer“ takes the value of the parameter „heat conductivity“ into account.



None

It does not interact nor does it move.

Static

It does not move nor does it take any damage. This is a nice behaviour for level boundaries.

Solid

It does not move (fall) on it’s own. It takes damage and heat. It is not displaced by other materials (no density check).

Move Like Sand

The pixel falls down due to gravity. If in contact with the ground it will move down or sideways in a 45 degree angle giving the impression of sand.

Move Like Liquid

The pixel falls down due to gravity. If in contact with the ground it will move down or sideways in a 90 degree angle giving the impression of a liquid. It will dipurse much faster than sand.

Move Like Gas

The pixel rises and disperses.

Combine By Errosion

The pixel combines with other pixels and can consume (replace) them in the process. The acid is a nice example of this.

Heat Transfer

The pixel transfers heat to surrounding pixels. Lava is a nice example of that. NOTICE: This is only needed if you want the pixel to GIVE heat. To receive heat this is not necessary.

Pixel Material Properties

Density

Density in Kg / m^3 (see: https://en.wikipedia.org/wiki/Density)
Examples:

- Steam: 0.6
- Air: 1.2
- Wood: 700
- Water: 1000
- Sand: 1600
- Granite: 2700

Friction

A value between 0 and 1. A value of zero feels like ice, a value of 1 will make it come to rest very quickly.

Gravity Scale

Scale on the gravity force. Snow for example has a lower gravity scale.

Bouncyness

A value between 0 and 1 that defines how much of the velocity is preserved if the pixel hits something and is reflected. CURRENTLY NOT SUPPORTED (may return soon).

Flow Speed

The maximum distance (in pixels per frame!) a material can flow in one step. You could view this as the inverse of viscosity.

NOTICE: It is in pixels per frame which is inconsistent with the other values that are in SI units. I have not changed it because it is only used in the simulation phase which operates on pixels.

Damage

How much damage this pixel is dealing to others (damage per second).

Self Damage

How much damage this pixel does to itself (damage per second).

Heat Conductivity

How good the material is in receiving heat energy. What ratio of heat it conducts from other pixels per second. It will reach the source pixels heat in one second.

Heat Sensitivity

How fast the pixel will loose health while burning.

Flammable

Can this material burn if the tempreature if above the ignitionTemperature?

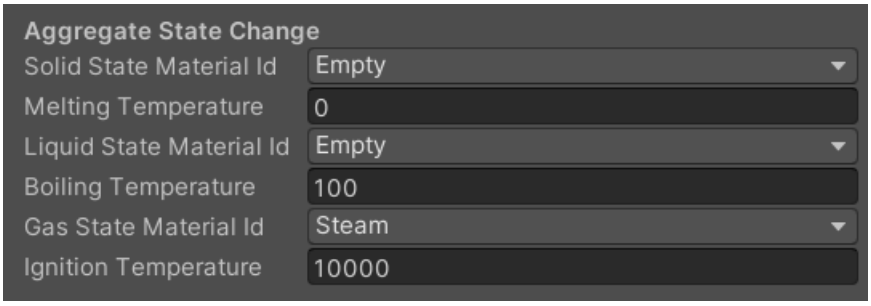
Pixel Aggregate Changes

If specified pixels can change to other pixel based on their temperature. In nature we have four different aggregate states: solid, liquid, gas and plasma. For simplicity this model does away with plasma and focuses on the remaining three.

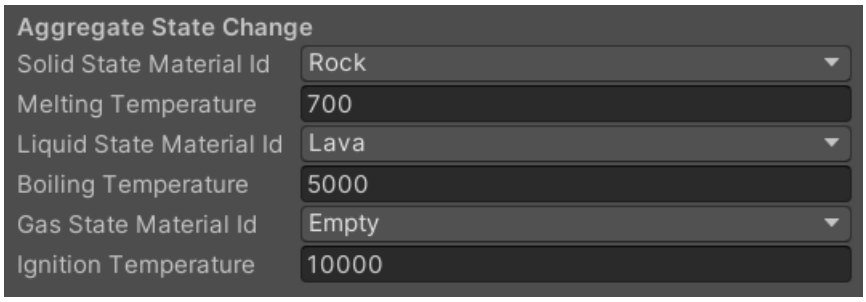
In the „Aggregate State Change“ section you can specify the temperatures and what will happen. Water can turn to steam for example. Or rock can turn to lava and back to rock.

The value „Empty“ means that this transition is not possible (nothing will happen).

Water → Steam



Rock ↔ Lava

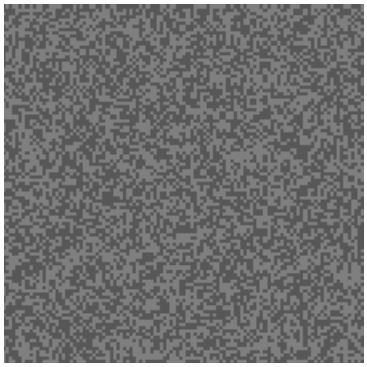


Pixel Colorization

At load time a color is assigned to each pixel. You can choose how this is done in the „Colorization“ section. Rock for example will be assigned a grey color which will then be darkened in 50% of all cases.

You can add your own color strategies in code, see PixelMaterial.SetPixelColors(..).

The result:



Level Building

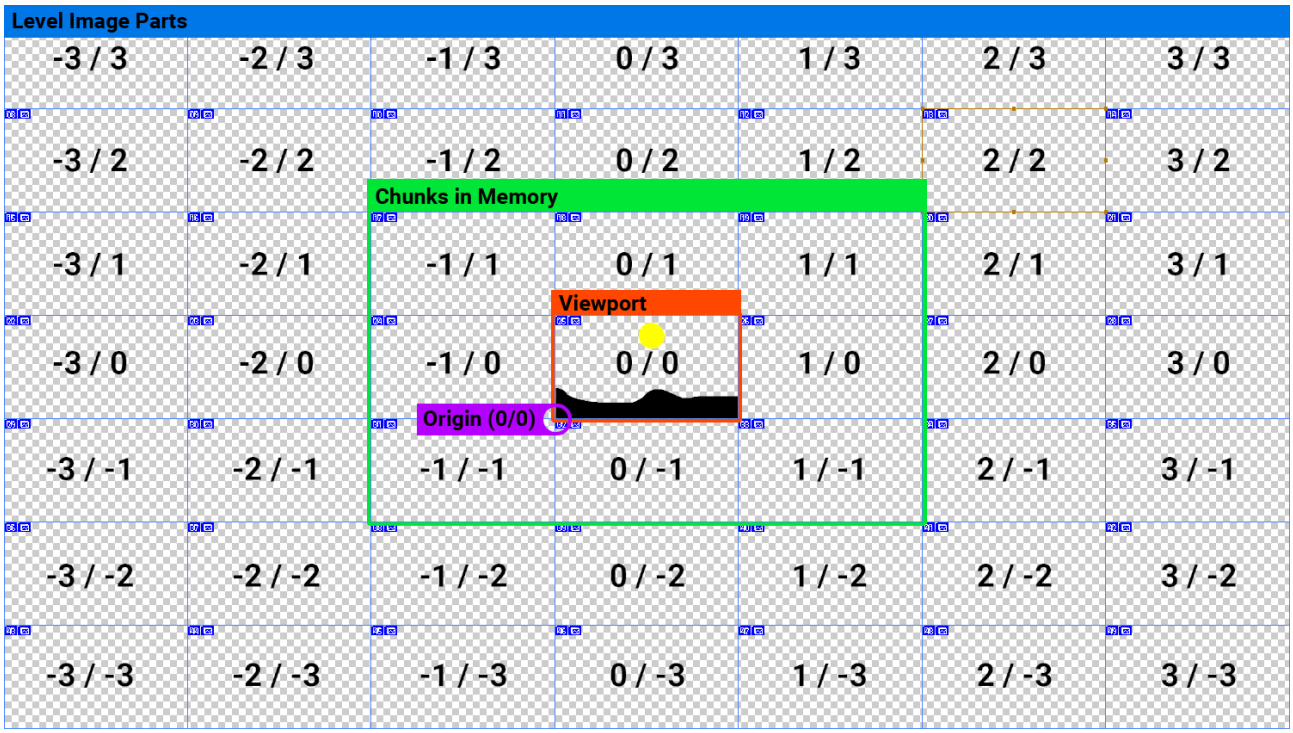
Since the pixel world is just a huge array of pixels you could edit and generate it programmatically. However, if you want to do some manual level design then an image based approach may be useful.

Level Loading (Level Chunks)

Conceptually the level is divided into 2D chunks of pixels. Each chunk will cover roughly one screen. At runtime these chunks are created from images in background as the player (i.e. the viewport) moves through the level. These image are called „level parts" and have a 1:1 relation to level chunks (one level part image per chunk).

HINT: Default level part resolution is 320x180. You can change the resolution but if you do then the demo levels will no longer load as they are made for 320x180. The level part image resolution has to match the resolution configured in your SandWorld game object.

The default start location for a level is the bottom-left corner of the 0/0 chunk. Initially only the surrounding chunks are loaded into memory. If the camera (i.e. the viewport) moves then more chunks are loaded into memory depending on what area of the level the viewport is looking at. This also means that the amount of required memory grows over time (chunks are not unloaded).



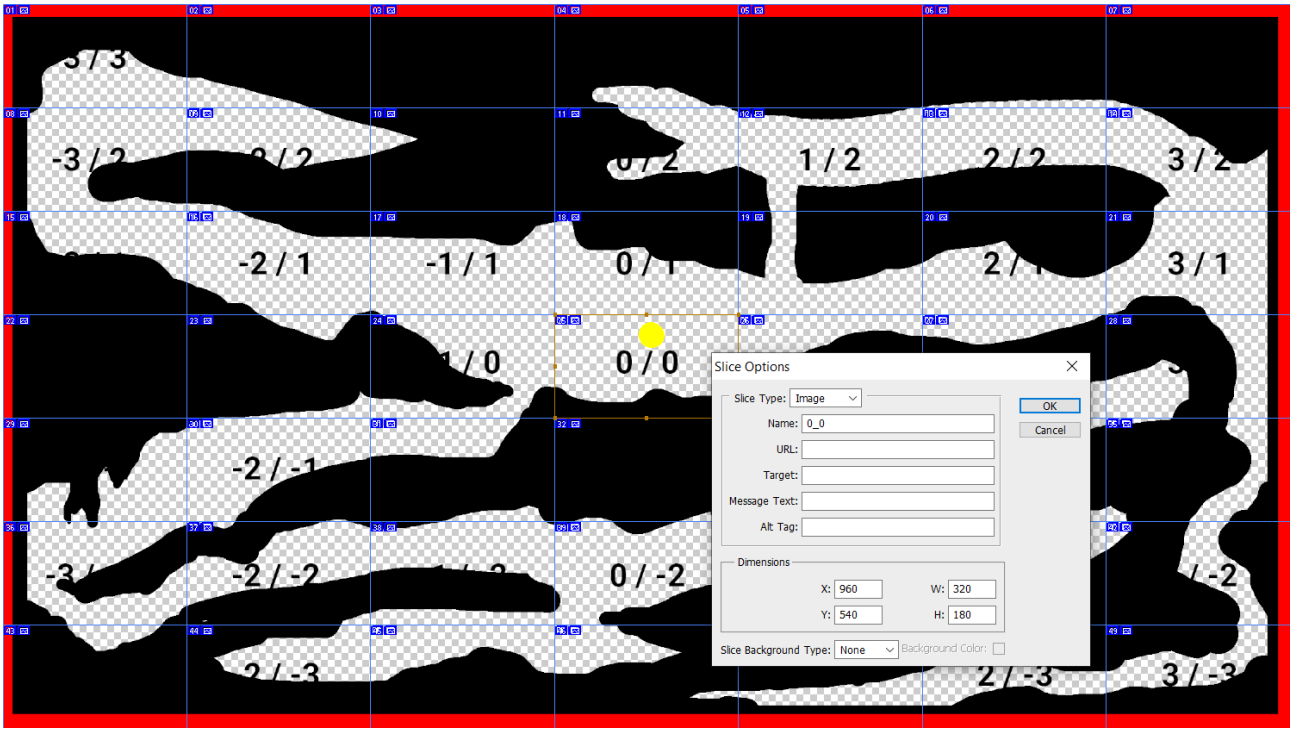
There is no limit on the number of chunks a level can have. The 7x7 grid from the demo is just an example. If a level part image is missing then it will be assumed to be empty and the corresponding level chunk will be empty too.

Each chunk is generated from an image (transparent png) and a some data (level info). Together they provide all the infos the game needs to create a new chunk in the PixelWorld.

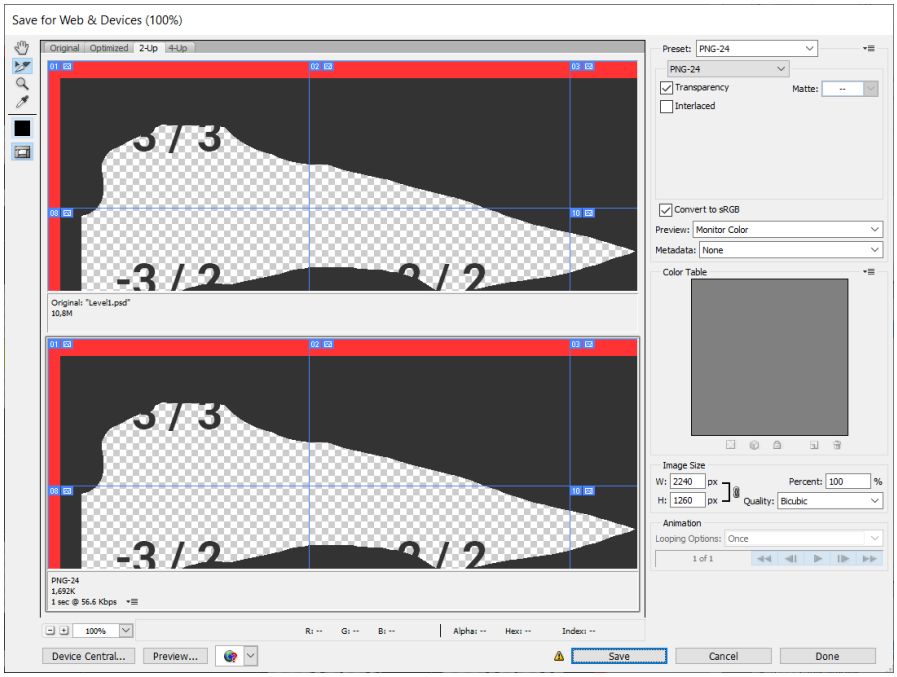
Level Generation (based on images)

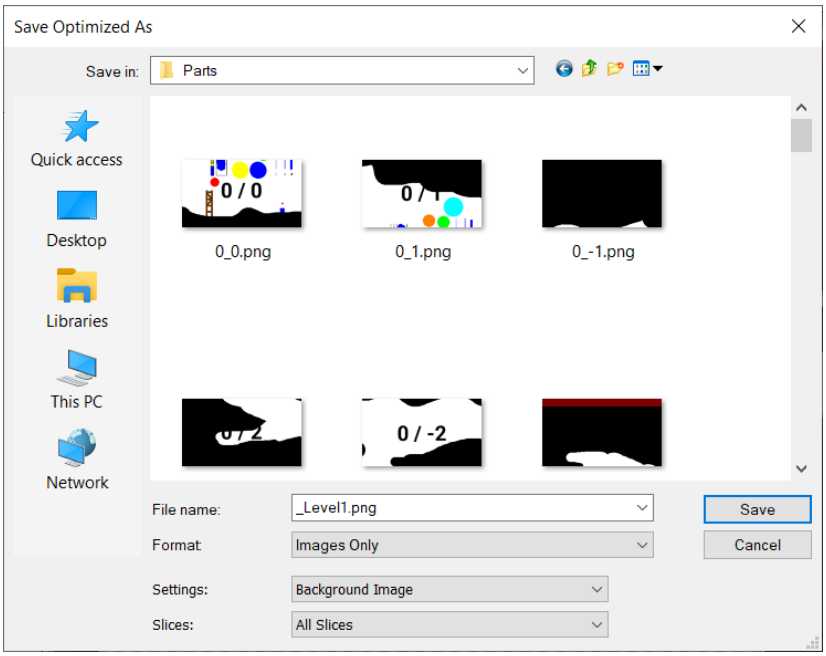
To create the initial pixel world we need a convenient way to place many pixels. Luckily for us there is something called an image file and image processors (like Photoshop). The template uses images (one image per level chunk) to build up the initial pixel world.

In the template you will find a folder called „Level1“. It contains one whole level that is already sliced up into level parts (one image per chunk in the „Parts“ subfolder). You will also find a „_Level1.psd“ file. Use this as a template for your own levels.

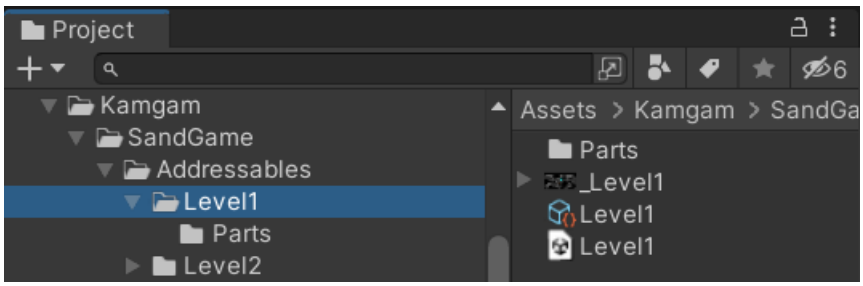


In this file the level is already sliced into 320x180 level parts and if you use „Save For Web ..“ then you can export these images directly into your Unity project folder („Parts“) as transparent PNGs.

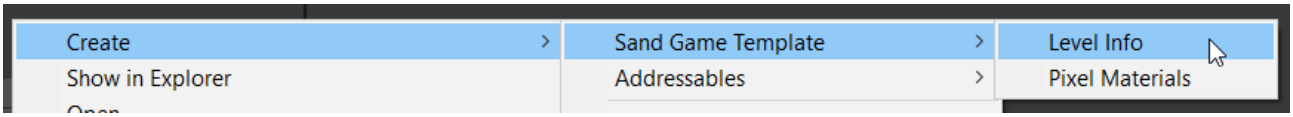




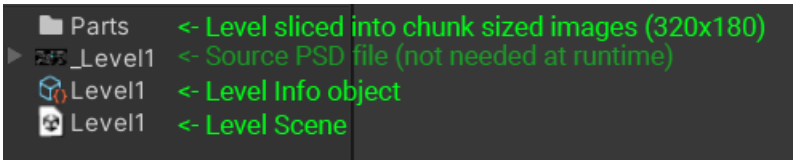
After exporting the images from Photoshop switch to Unity and find (or create) the „LevelInfo" object. In the Level1 and Level2 demo level these already exist:

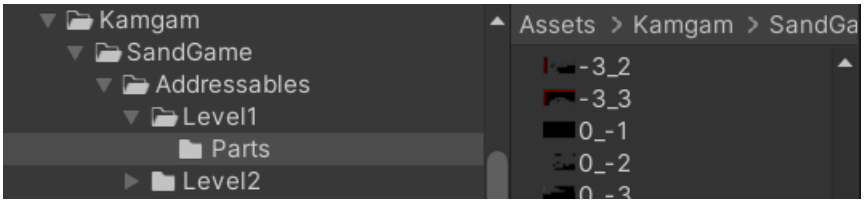


For new levels you can create them via „Right Click > Sand Game Template > Level Info"



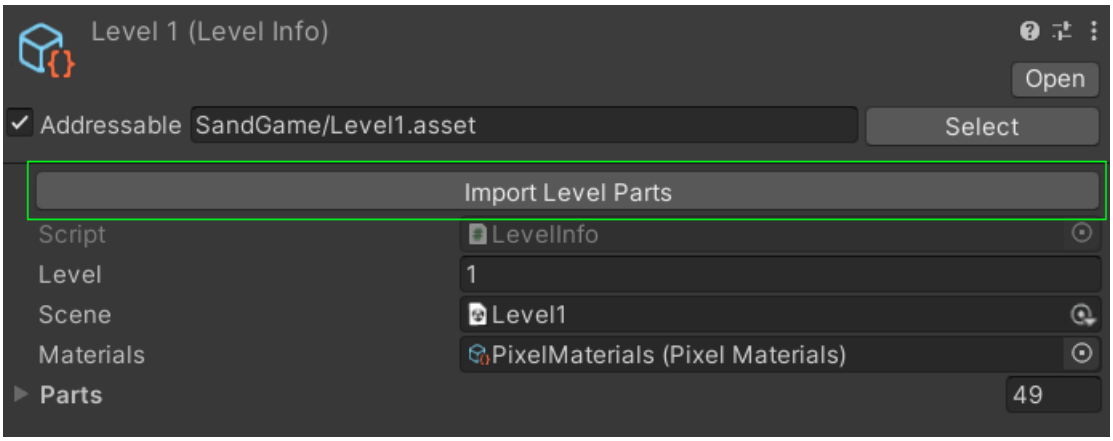
For each level no need to provide these files:





Once you have the image parts exported you need to add them to the level info object.

Since doing this manually is cumbersome the info object has a button to do it autoamatically called „Import Level Parts“

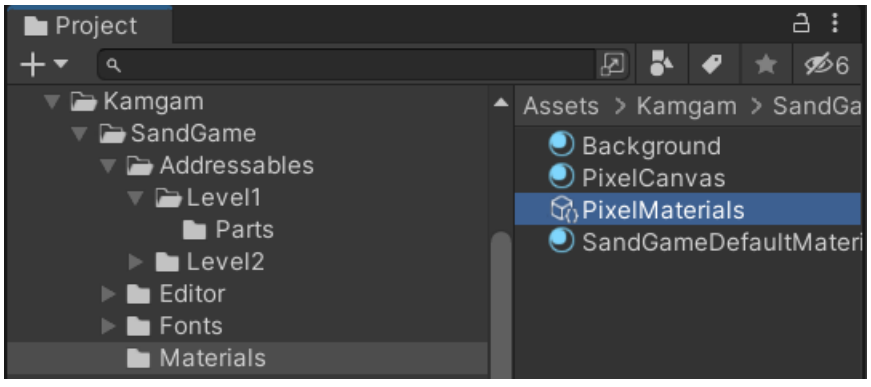


If you click the „Import Level Parts“ you may have to wait a bit while the editor prepares the images. It will make them addressable (Unity Addressables System) and set the format options.

After importing you should see the parts in the level file, like this:

Don’t forget to make the level info object addressable and choose a path conforming to the level info naming convention: **SandGame/Level#.asset** (see LevelInfo.StaticAPI → GetAddressablePath() in code). You’ll also need to specify a level number, level scene and add a reference to a PixelMaterials object.

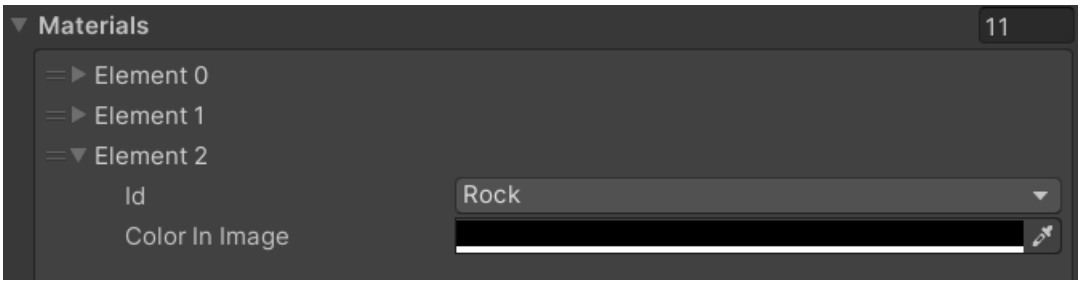
Pixel Materials for Level Loading



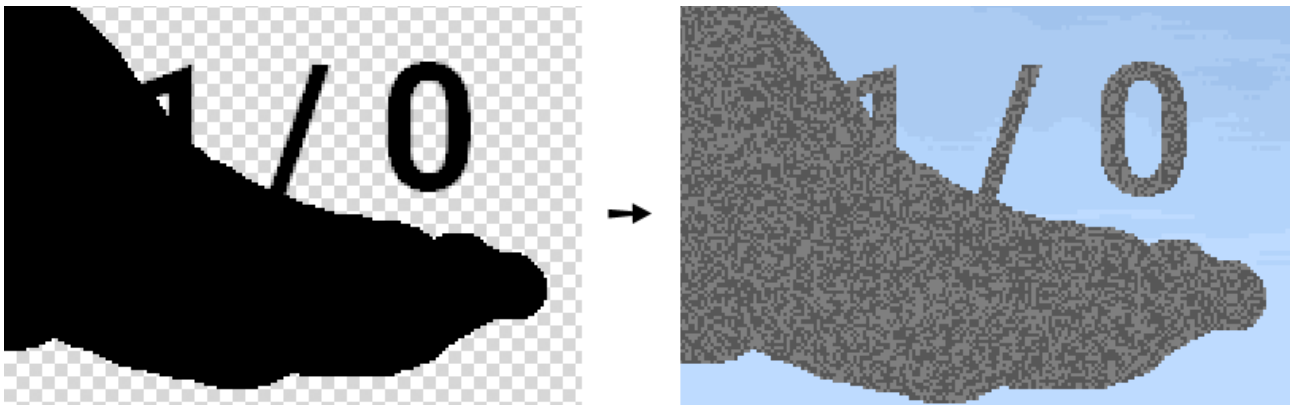
Basically it works like this: **Color in Image = Pixel Material**

To do this conversion the system needs a lookup table (what color = what pixel material). The Pixel Materials contains that info because for each material there is a „Color In Image“ property.

Rocks for example have the color black.



This means all black pixels in the level image will be converted to rock.



HINT: If you look into the code then anything that starts with „LevelPart" is ONLY about LOADING the images. Anything that starts with „PixelWorld" is about the actual pixel world

At load time a color is assigned to each pixel. You can choose how this is done in the „Colorization" section. Rock will be assigned a grey color which will then be darkened in 50% of all cases.

You can add your own color strategies in code, see PixelMaterial.SetPixelColors(..).

Performance

In this section I describe the most performance critical properties of the simulation and how you can use them to increase the performance.

Simulation Buffer Size

PixelWorld.SimulationBufferSize 2 by default

How big the simulation buffer rect should be in relation to the screen size (2 = double the size, 3 = tripple ...). Increasing this costs A LOT of performance (grows x^2). Don't do it unless absolutely needed. If you increase it in one direction think about reducing it in the other.

Simulation Frame Rate

PixelWorld.FrameRate, 60 by default.

The simulation is optimized towards 60 frames per second. If you change this too much you will see diverging speeds between simulated pixels and not simulated content (free-falling pixels, the Unity scene objects, ..).

If „LimitToApplicationFrameRate" is disabled then the simulation frame rate is independent of the application frame rate. It sets the frame rate at which the simulation will run (it may run 0, 1 or more times per frame, just like fixed update).

However, if the simulation frame rate does not match the application frame rate then the timings (Time.deltaTime and PixelWorld.FixedTime) will diverge.

HINT #1: Use the Simulation Frame Rate slider on the PixelWorld for debugging. It helps a lot with analyzing what the pixels are doing.

If possible keep the the simulation frame equal to the apllication frame rate (LimitToApplicationFrameRate = true). If you use a custom frame rate then try to make it a multiple of the application frame rate to ensure the timings match up.

If you have code that relies on the simulation frame timing then make sure „MatchApplicationFrameRate" is enabled or that you compensate for FixedTime deltas in your code.

If you have some objects in the Unity scene that you want to move in sync with the simulation frame rate then you can use this delegate:

```
PixelWorld.OnPixelWorldFixedUpdate(PixelWorld pixelWorld)
```

It is called 0, 1 or more times in Update(), right before the simulation but after frameCount, time etc. have been increased. You can fetch the timing properties (Time, FixedTime, ..) from the pixel world to sync your animations with the pixel world.

Limit To Application Frame Rate

PixelWorld.LimitToApplicationFrameRate, true by default.

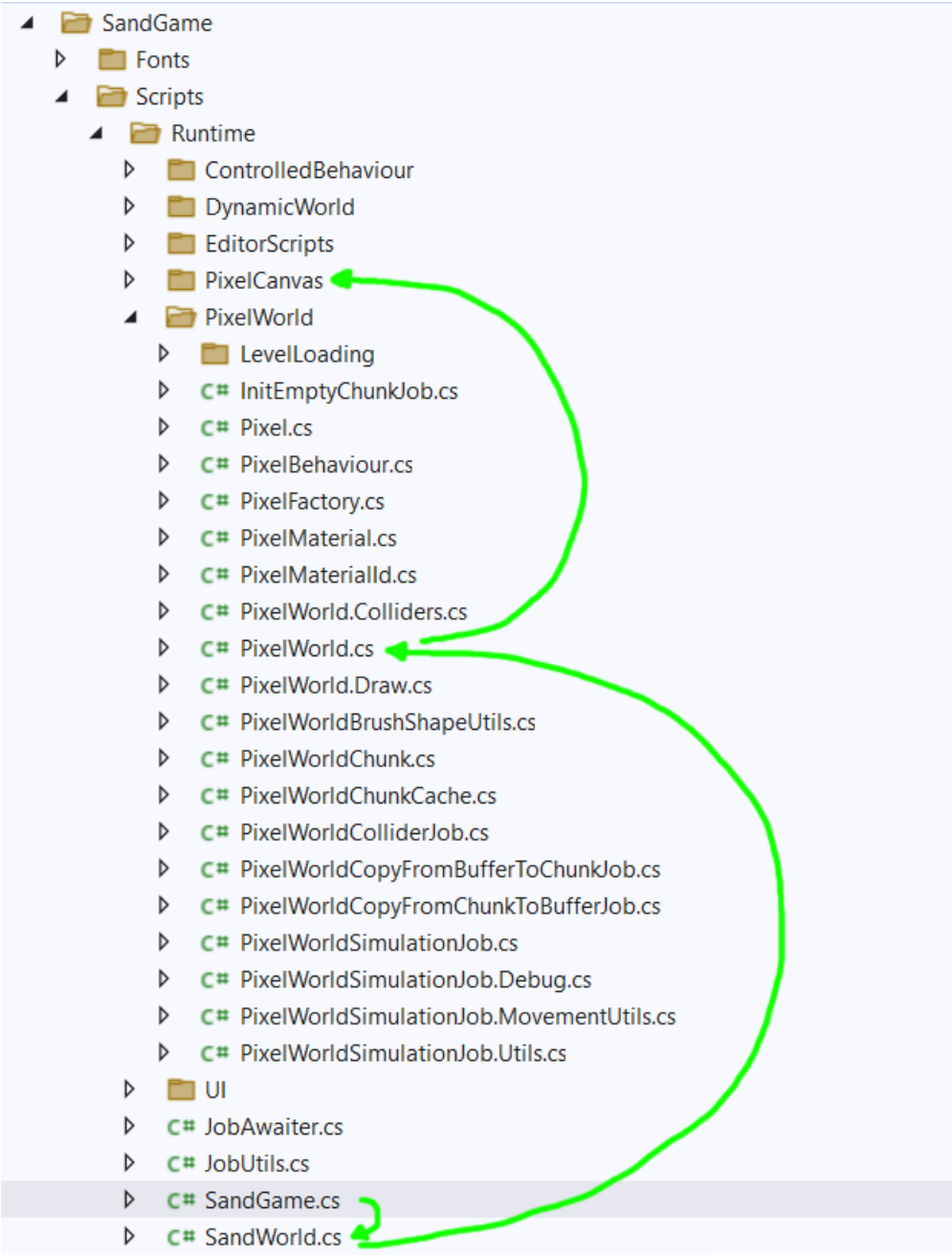
If enabled then the simulations actual simulation frame rate is capped to the real application frame rate (i.e. the simulation is executed at most once per frame). If the fps drop then so do the simulation fps. This is useful to keep your simulation step rate and the actual frame rate in sync.

Details / Extending the Template

To extend the template you have to understand the underlying code. Sadly there are no GUI options for dealing with BURST and Jobs. The following section will give you an overview of how the code is structured and where to look to extend it.

Code Overview

SandGame → SandWorld → PixelWorld → PixelCanvas



The base class of the template is the „SandGame“ class. This one loads the menu etc.

Though the pixel stuff only starts with the SandWorld class which itself is just a convenience wrapper around the PixelWorld class which is where the heavy lifting is done.

The pixel canvas is basically a wrapper for a RenderTexture.

Simulation Job

The simulation job is where most of the work in terms of pixel behaviour is done.

It's divided into separate partial classes:

C# PixelWorldSimulationJob.cs
C# PixelWorldSimulationJob.Debug.cs
C# PixelWorldSimulationJob.MovementUtils.cs
C# PixelWorldSimulationJob.Utils.cs

Debugging:

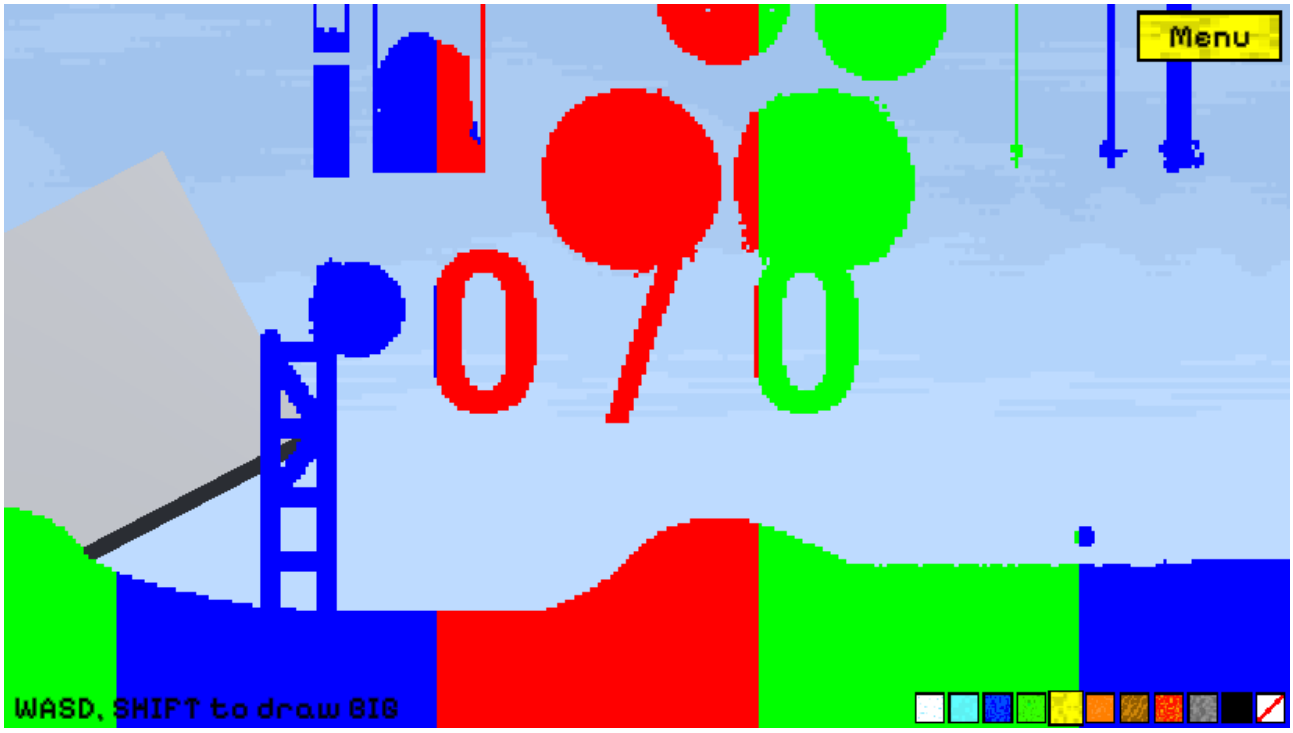
If you look into the simulation job then you will find these lines:

```
#if UNITY_EDITOR
    // Debug pass (only enable if you want to debug something)
    debugPass();
#endif
```

These refer to the first method in „PixelWorldSimulationJob.Debug.cs" where you can uncomment whatever debug drawing you need.

NOTICE: The debug drawing methods do change the pixel colors permanently. Thus reverting back to the default colors after debugging at runtime is not possible.

Here for example is the debugColorizeByThreadNr() method. It adds a color to each simulation thread. As you can see each thread covers a vertical slice of the simulation buffer.



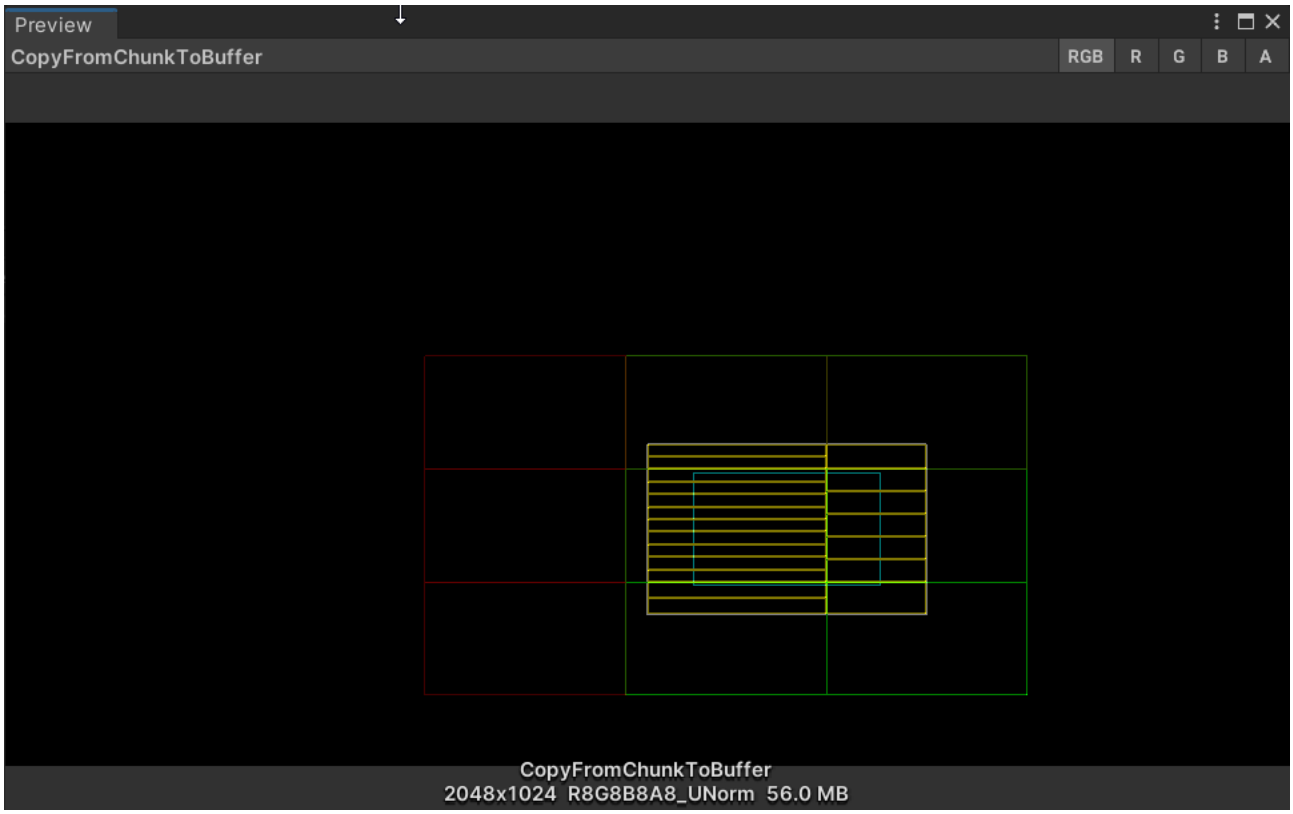
What's with all the ref NativeArray<PixelMaterial> in the simulation code?

A pixel is only fully defined by the data inside the pixel struct (see Pixel.cs) and the pixel material properties stored in the PixelMaterials object. That's why we pass around a reference to the materials in many places (basically anywhere the material property infos are needed).

Level Chunks / Chunk Jobs

At runtime the level is divided into level chunks. Each chunk is a big array in memory representing a rectangular part of the level. If the viewport moves then new chunks are created in memory and the level info object is checked for possible images to load into that chunk. The image loading can take multiple frames. That's why new chunks are always initialized with empty pixels immediately (InitEmptyChunkJob) and only later they are filled with the actual pixels based on the image.

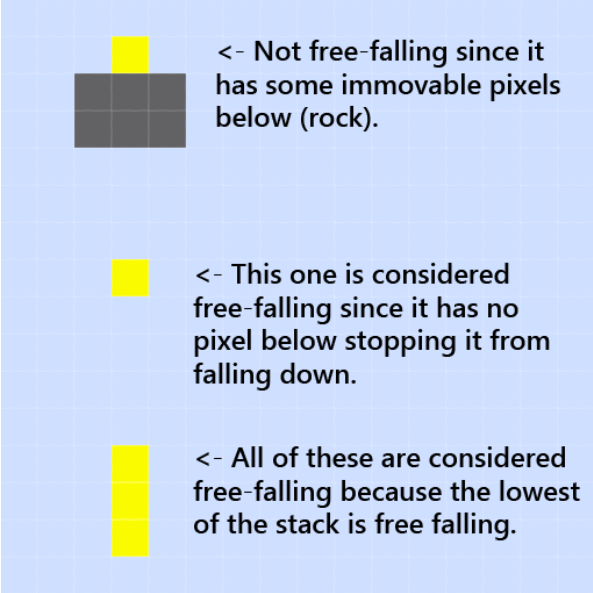
red = Chunks that are loaded but not needed
green = Chunks that are loaded and copied to / from the simulation buffer
white = the simulation buffer (barely visible around all the yellow threads)
yellow = the copy pixels from chunks to simulation buffer threads
teal (inside the white/yellow area) = the viewport (aka what the player actually sees)

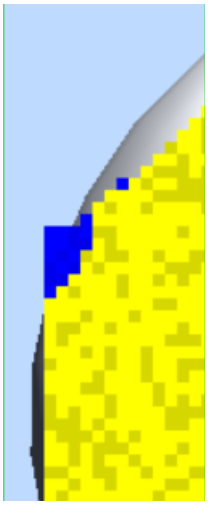


About free-falling pixels

In the code we differentiate between pixels that have solid pixels below them and pixels that can fall down. Those falling pixels are called free-falling.

At the end of each simulation step the free-falling flag is updated. Free-falling pixels can form vertical chains (if all pixels below the current pixel are free falling then the current pixel is also free falling). This is done to make the logic of pixel movement a bit easier and give a better (more natural) falling and flowing behaviour.

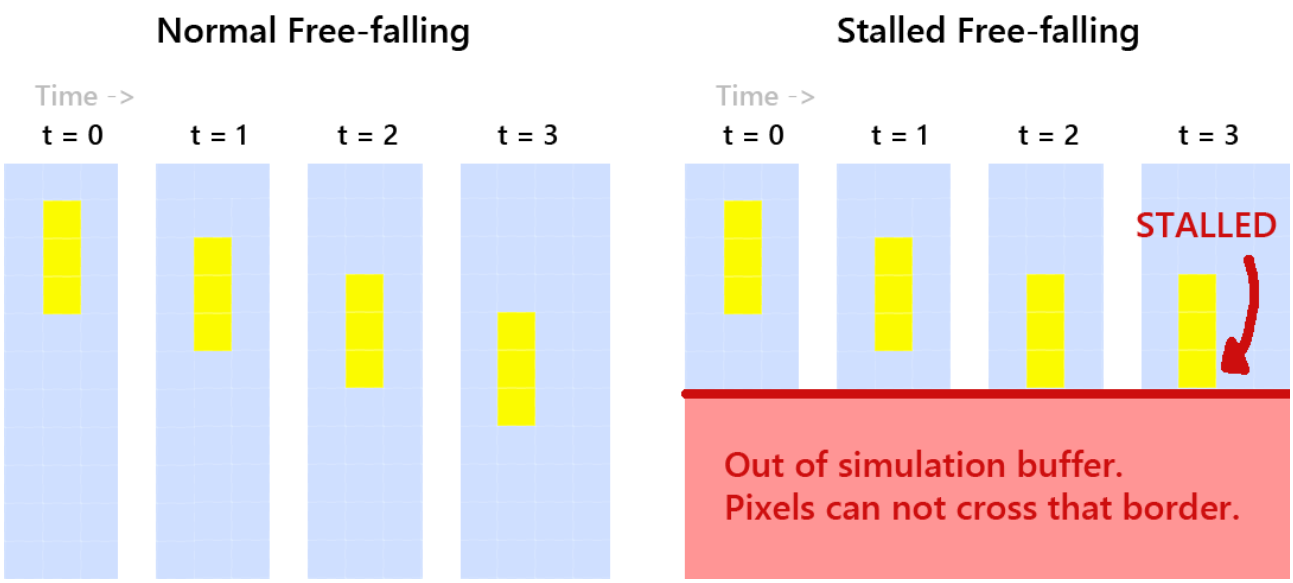




Now what does free-falling have to do with the level boundaries and the „frozen" pixel effect you can sometimes see?

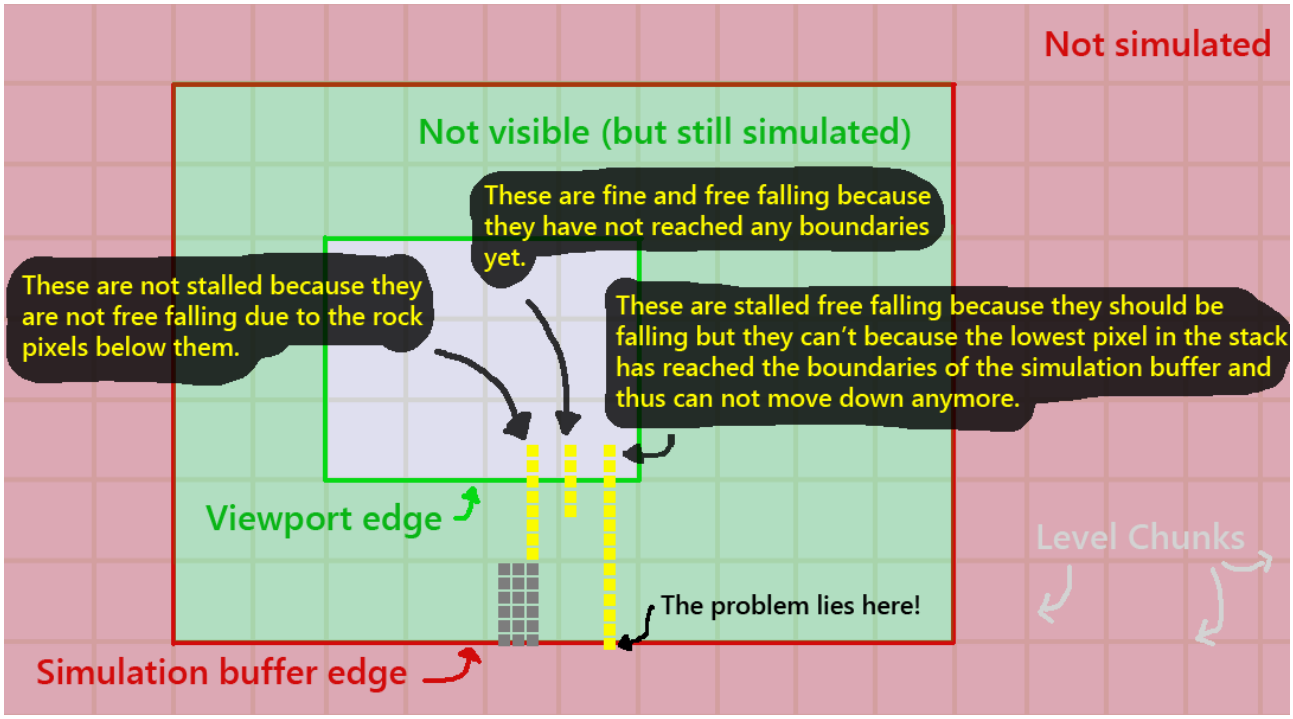
← Notice the empty areas on the left which should be filled by sand and water flowing to the left yet sometimes they do not. That's because they are stalled free-falling (details below).

The thing is if free-falling pixels hit the border of the simulation buffer then they are stalled. They are still considered free-falling by the code but since there is no space below them in the buffer they can not move downwards.



The thing is stalled free falling pixels will not move sideways (because they are still supposed to move downwards) BUT they can not move downwards because they have reached the simulation buffer end. That's what you see in the level 2 demo.

The thing is stalled free-falling pixels will not move sideways because they are still supposed to move downwards. They can not move downwards because they have reached the simulation buffer end. That's what you see in the level 2 demo if you let it run and look at the bottom left corner.



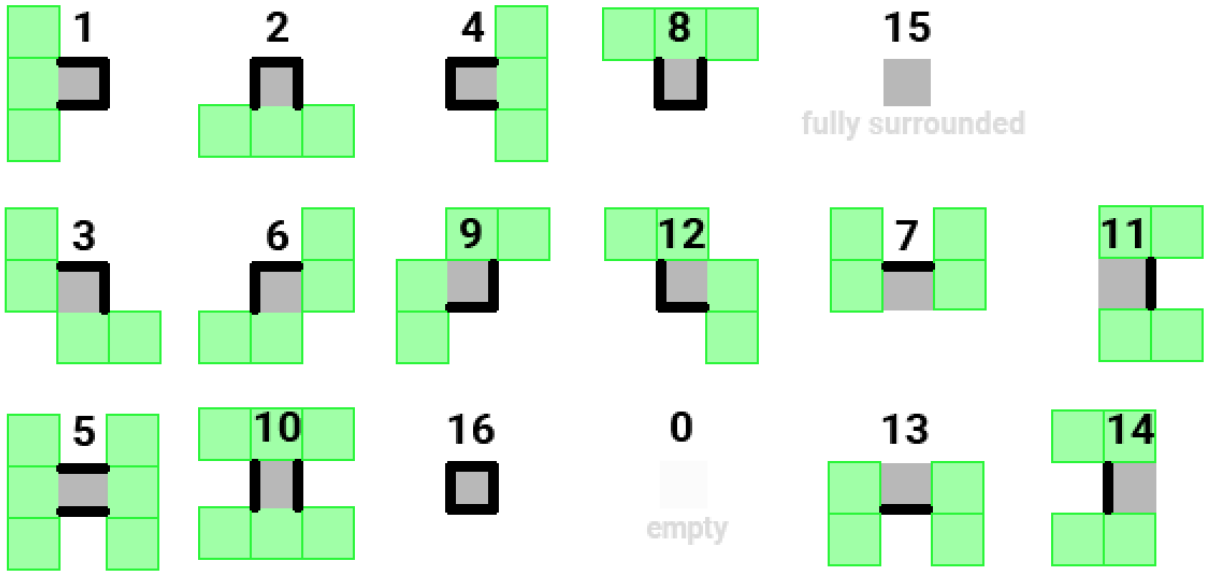
BURST Caveats

Do NOT use „bool" types in BURST code. It works in the Editor but it will fail in builds with a „ArgumentException: Object contains non-primitive or non-blittable data." (source). Instead we use a byte and declare 0 = false and 1 = true.

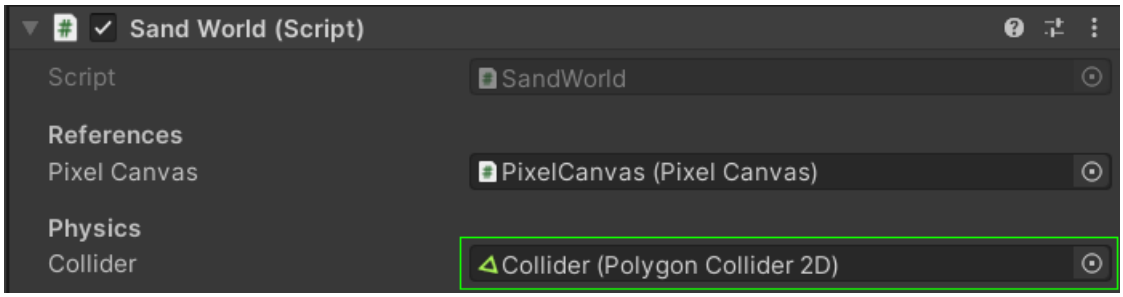
Collider Job

The collider job visits each pixel an categorizes it (assigns a code from 0 to 16). This is often referred to as a „marching squares" algorithm.

Below you see path pixel categorization codes and possible next path positions (green). Possible path positions are checked in counter-clockwise order.



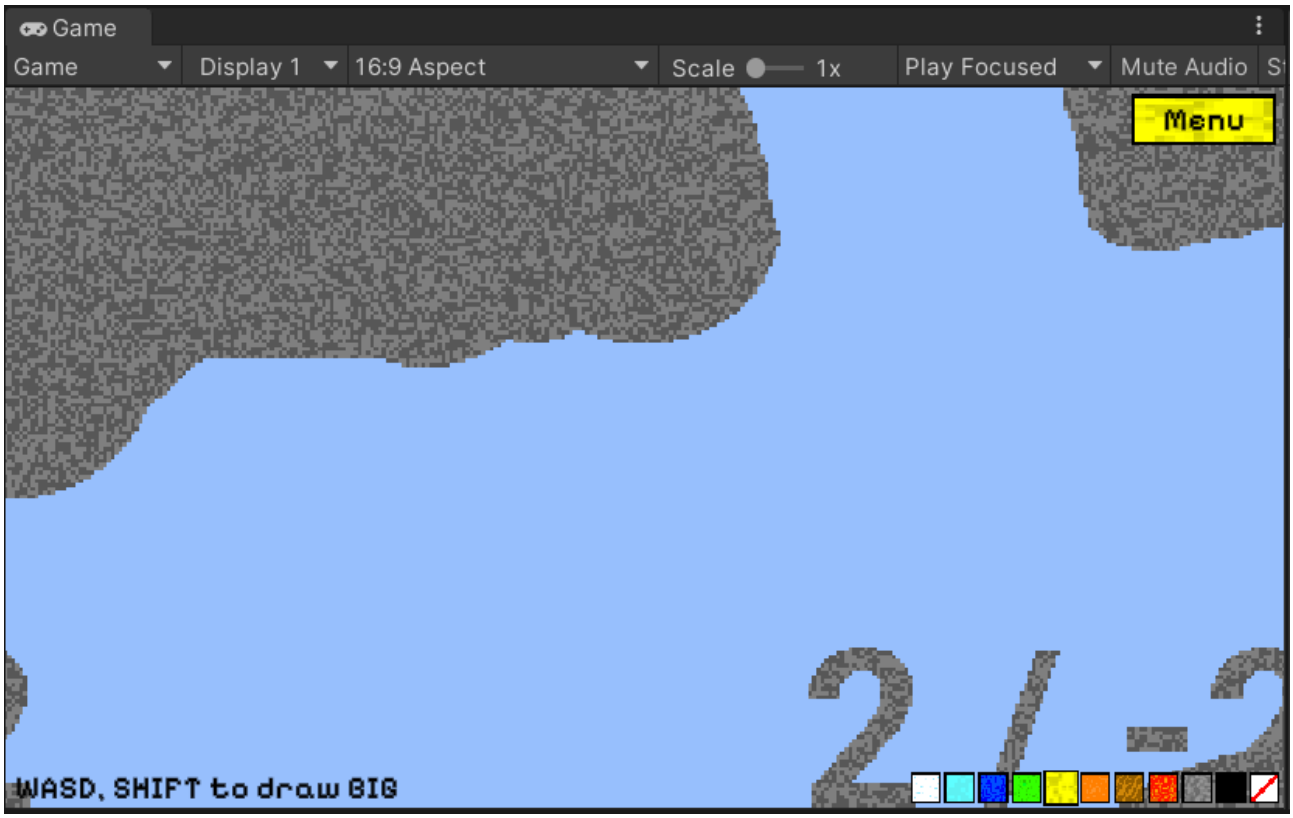
After the collider creation process you will have a code for each pixel and a list of paths. The „PixelWorld.Colliders.cs" class uses that information to update the PolygonCollider2D that is referenced in the SandWorld object.

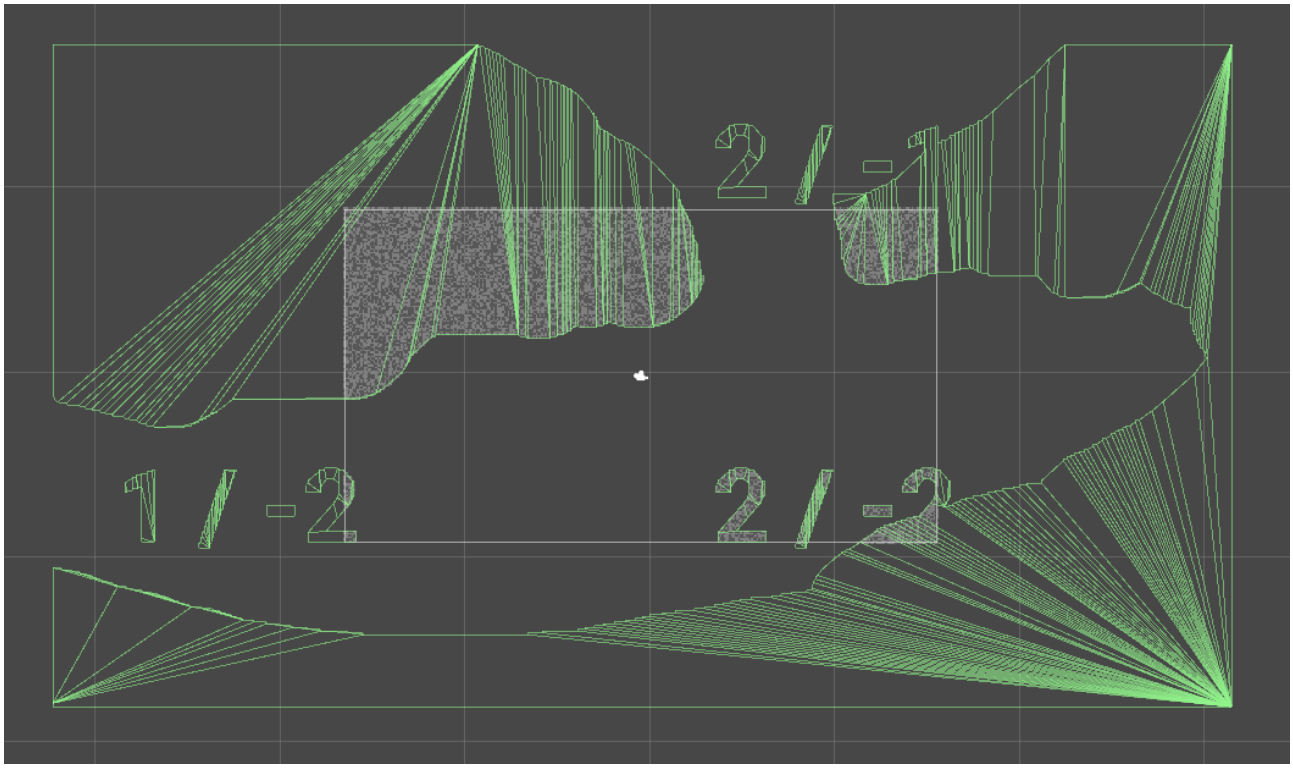


In the end it also cleans (merges) all collider vertices on straight lines (x, y, diagonals) to reduce the vertex count.

HINT: Use the collider information to do stuff in your Unity scene based on level geometry.

Keep in mind that the collider is ONLY generated for a limited area around the viewport:



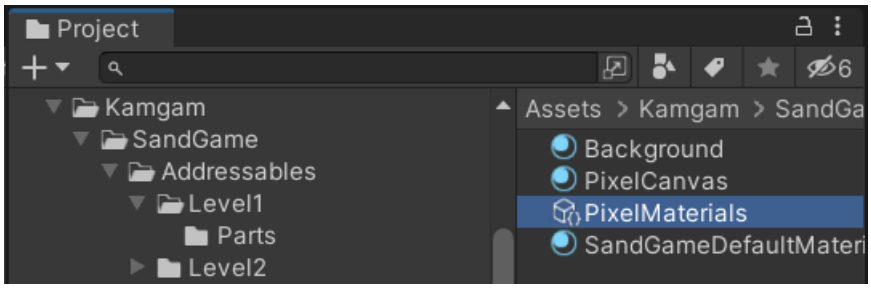


If you have physics objects in your Unity scene that rely on collision with the pixel world collider then you will have to take care of them once they are outside the collider area surrounding the viewport (i.e. they will start falling through the floor once the collider is gone).

How to add new pixel materials

1) Pixel materials are compared by their material id, so you will have to look into „PixelMaterialId.cs“ and add a new id.

2) Then add the new material to the „Materials“ list in the „PixelMaterials“ asset and configure the material properties.



And that’s about it. Now you should be able to use your new material.

Frequently Asked Questions

Here are some common issues that have been reported.

Unity 2021.3 or higher is required. If you can, please upgrade to the highest LTS version of Unity. The newer the version the less „glitches" it usually has.

Why only 320 x 180 pixels per screen?

Two reasons:

First, creating levels with high pixel density is a real chore and they tend to be very hard to design. If you look at games like Noita you can see how much you can actually do with a low pixel resolution. If you count the pixel in the image below you will get a width of approximately 300 pixels. Low resolutions help you with keeping your game design from falling into chaos.

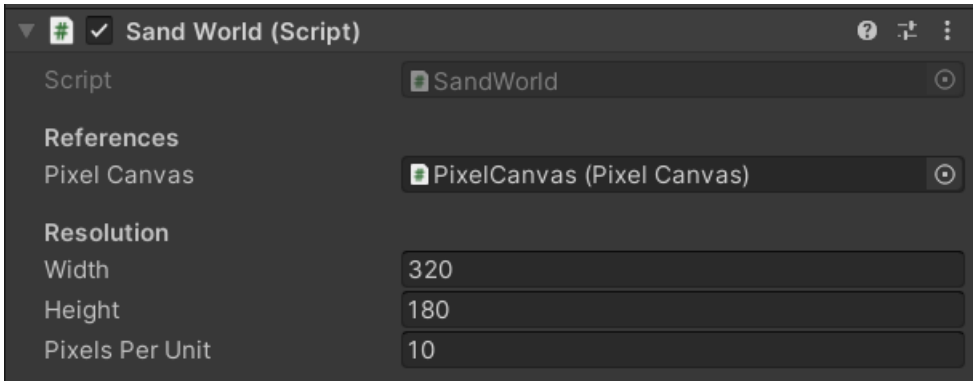
Second, the less pixels you have the more elaborate your simulation can be. Each pixel needs to be simulated so the number of pixels plays a crucial role in performance. Even if everything is multithreaded.

You'll have to strike a balance between performance (pixel resolution) and your game design needs.

There are much fewer pixels in Noita than you would think.

How do I change the pixel simulation resolution?

Simply edit them in the sand world inspector.



Please notice that you will have to generate your own level part images matching that resolution. The default demo level parts won't load with resolutions other than 320x180.

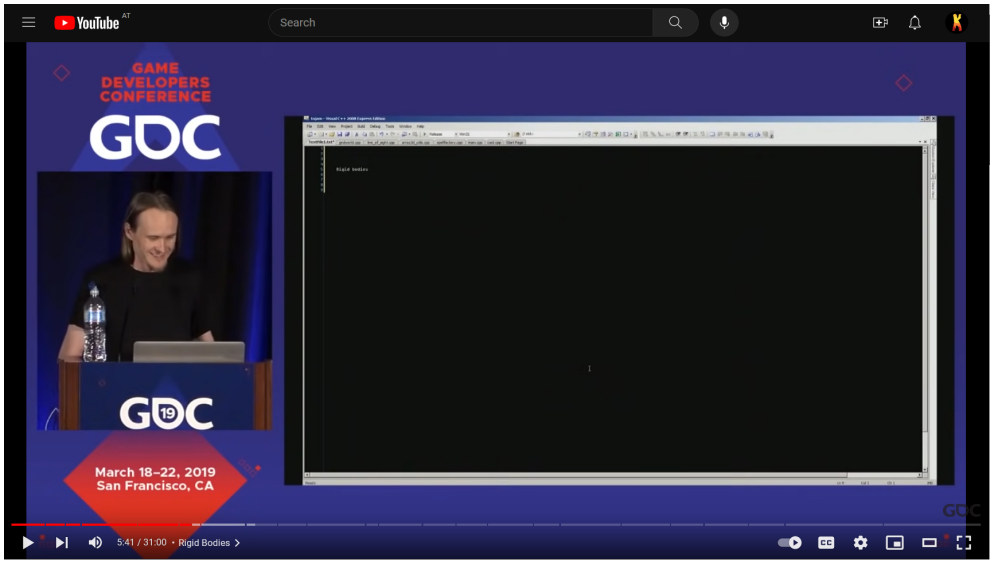
What things do I need to implement myself?

As stated in the intro this is just the base for a falling sand game. If you want to make this into a more sophisiticated game (like Noita) then there is plenty left for you to do. Most notably:

Rigidbody Physics

The template does not include support for rigidbody physics. Though this is one of the features I would like to add if the template gets popular enough (i.e. it earns enough so I can justify the time investment).

To get you started I'd recommend you watch this GDC Talks by the creators of Noita:
https://youtu.be/prXuyMCgbTc?si=KaT-JFaPX-jo4G3-&t=341



Basically what you'll have to do is to convert your pixel object into a 2D physics shape (triangulate the pixel outline). Then you simulate the rigidbody and then you copy it back at the positions to the pixel world. I have mentioned the timings for physics in the „Frame Lifecycle" section. Look for „DynamicObject.cs" in the code to get started. - This is one of the first things I will add should the asset prove to be popular.

World persistence and chunk unloading

By default the world is loaded (streamed) at runtime based on the viewport position. The crux is that at the moment these loaded chunks are never unloaded, meaning the memory consumption will increase with every new part of a level. The memory is freed once you unload a level. If you need really big seamless level then you would have to implement some unloading of level chunks.

You basically have two options:

A) Unload the level parts that are the furthest away from the viewport. Caveat: Since you are unloading a level chunk all the changes the player has made to the pixel world in that chunk will be lost. If the player then enters that part of the level again it would reload it based on the image part PNG and thus the level would look like it never changed. Depending on your game design this could be okay but it probably is not since messing around with the pixel world is a big part of the fun in these kind of games.

B) Unload the level parts that are the furthest away from the viewport BUT save the chunk state to disk before you do. This will require you to implement a multi-threaded serialization and deserialization of the whole level chunk state. Not a trivial task. If you want to approach this then start with the „`PixelWorldChunkCache`" class since that's where the current in-memory caching of existing pixel world chunks happens.

Why does this not use ECS (Entity Components)?

The reason is simply memory layout. Sadly, there is no way (yet) for ECS to use a specific memory layout. If I add each pixel as an entity then iterating over an area of pixels (x/y coordinates) would mean randomly accessing entity components (that's „slow").

In my own system I can control the memory layout and have the pixels saved just like in a RGBA32 texture (one row after another). This allows me to immediately calculate the index of a pixels location in memory and thus it is more efficient when iterating over millions of pixels. For pixel material access (random access) the performance should be roughly the same or ECS would be even faster. I have not yet profiled this.

Could the simulation be moved to the graphics card to speed it up?

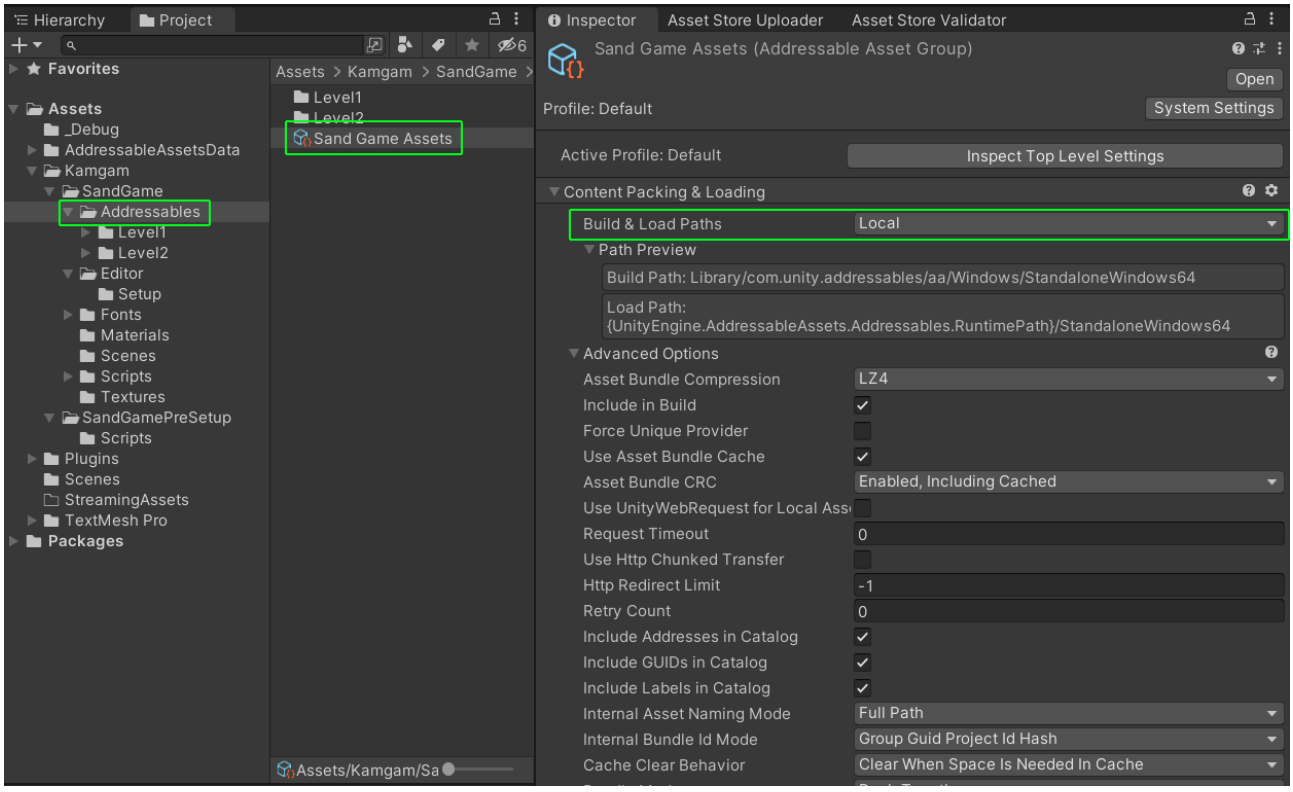
That's a tricky one. If it was a double buffered cellular automata then I think it could be done with a compute shader. Though this simulation isn't. It writes the result of each pixel simulation directly into the buffer that is used as the input for the next pixel. This has some advantages in terms of „realism" in the simulation but at the same time it makes parallelization more difficult.

If you moved the current simultion to a graphics card you would have to constantly share state (the whole buffer) between the parallel execution units. CAVEAT: I am not a expert on that matter so maybe there is an easy way which I don't know of.

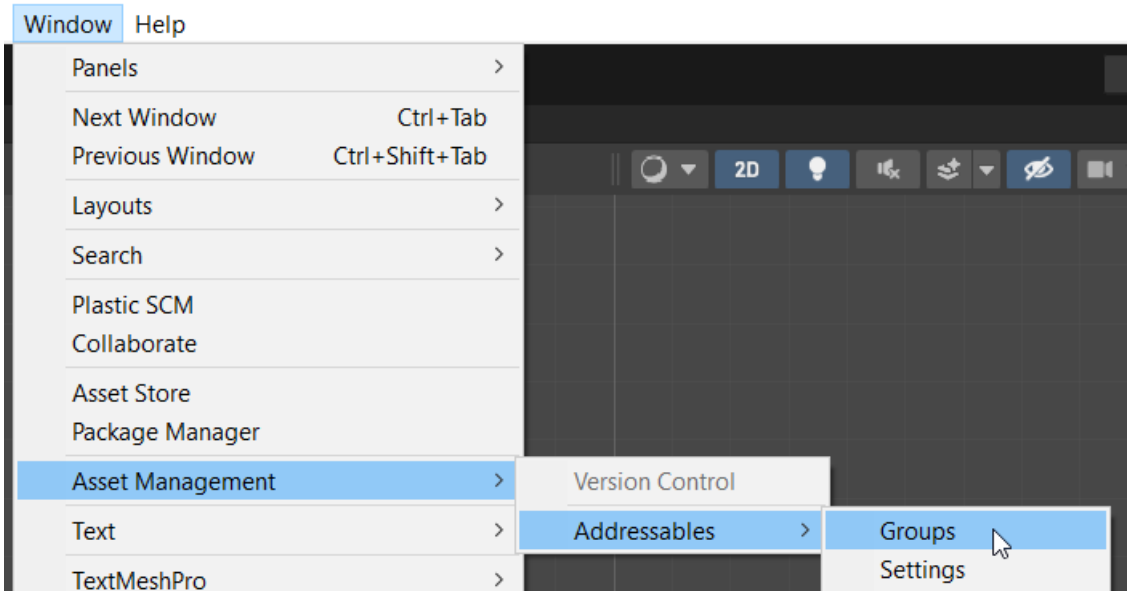
I get some sort of Addressables error during the build. The build succeeds but then the levels do not load?!?

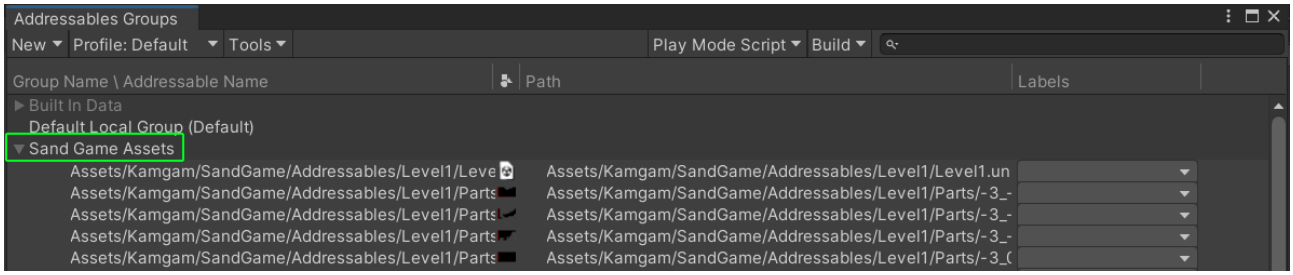
This seems to happen sometimes (am investigating). If you see this happen please check two things:

First: please check that the „Sand Game Assets" addressables group is set to „local" or whatever setting you need for your setup. It sometimes resets to „<custom>" upon import.



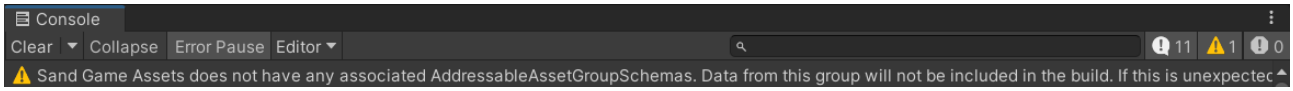
Second: Please also check that the „Sand Game Assets“ group is in the addressables group list. If it is not then please add it and check if the group actually contains all the assets.



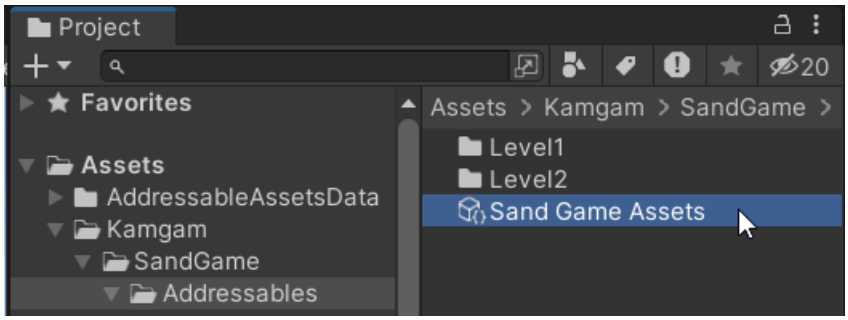


When building Unity complains about a missing group schema on „Sand Game Assets“.

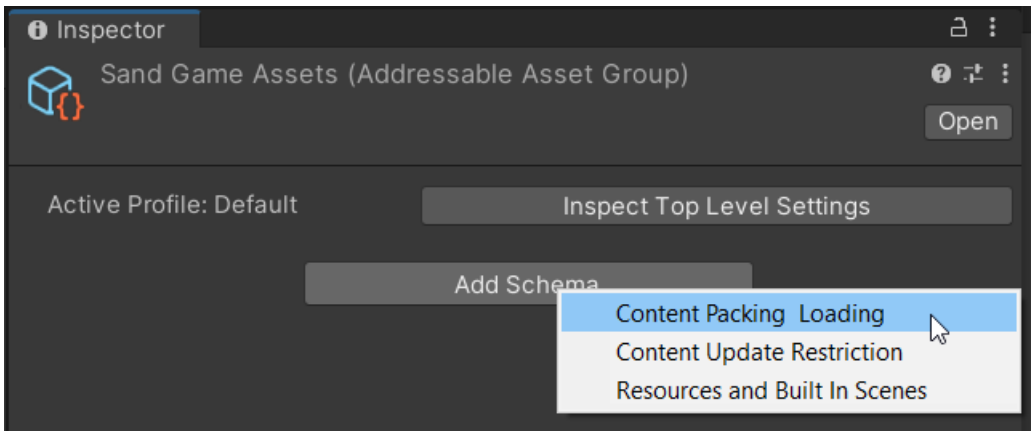
Depending on your Unity version you may have to add a schema to the „Sand Game Assets“ group asset.

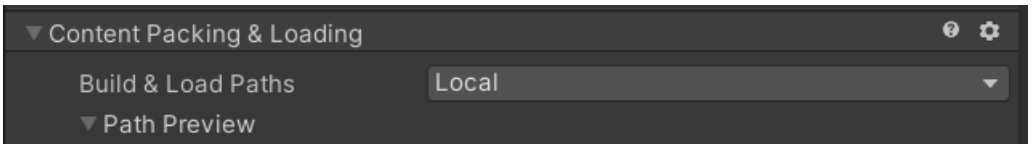


To do this please find the asset and add a schema that fits your needs.

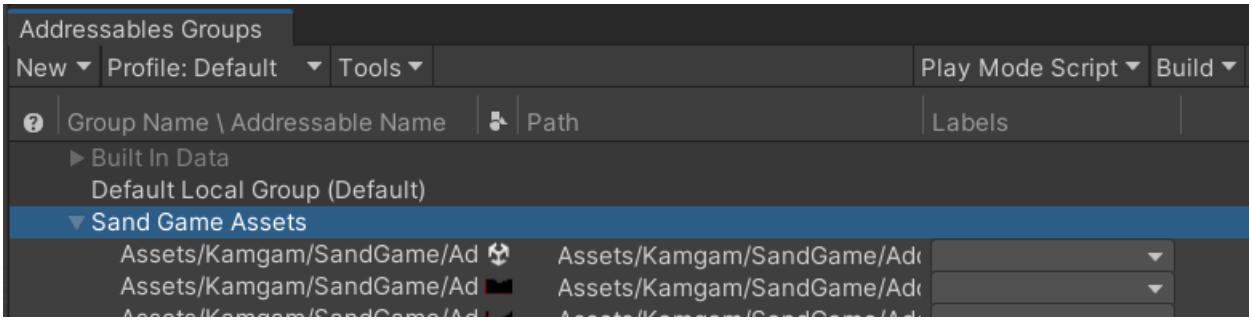


Usually adding a „Content Packing Loading“ is enough to make it work.

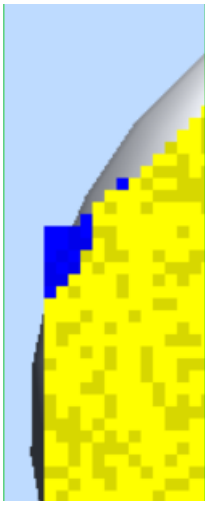




Please also check that the asset group and the assets within are listed in the addressable groups (Window > Asset Management > Addressables > Groups):



I see some pixels hit a boundary in the demo level 2 (bottom left corner)



← Notice the empty areas on the left which should be filled by sand and water flowing to the left yet sometimes they do not. That's because they are stalled free-falling (details below).

This is to be expected and a normal side-effect of the simulation buffer size.

What you see there are stalled free-falling pixels. Please refer to the free-falling pixels section above for details.