Kwyll User Guide

Paul Gregory

2026-05-02

Table of Contents

The Retro 8-bit Game Creator

What is Kwyll?

If you're new to Kwyll, a quick introduction to what it is, why it is, and how it came about would probably be useful, so here goes.

Kwyll is, at it's core, a tool for creating retro-styled games for retro platforms. What is "retro"? Well, in terms of Kwyll, it will initially include anything that existed at the start of the home computer revolution, so primarily 8-bit platforms, very low power, simple capabilities, such as the Sinclair ZX Spectrum, Commodore 64, Amstrad CPC, etc. It may extend beyond that in time, but that's the primary target. In its current form, Kwyll only targets the Sinclair ZX Spectrum (my personal favourite machine from the era).

That covers what it is, why it is, well that's a longer story, but I'll try to keep it as short as I can. I've been involved with computers and software development since I was 14 years old (1983!), and my first, well actually second, my first was a ZX81, computer was a Sinclair ZX Spectrum, the 16K version which I later upgraded to 48K myself when I got the upgrade as a Christmas present from my parents. I've spent a good portion of my career working in games, starting on the 8-bit platforms of the time at Incentive Software. As such, these platforms hold a great deal of nostalgia for me, memories of a time I've always been very fond of. Fast forward to recent times, after a career working in the software industry, with some very complex systems including visual effects, VR, modern video games systems, FinTech etc. I found myself drawn back to the early days, when things were simpler and more fun, well, that's how I remember it anyway. A new wave of respect for retro tech. in all shapes and forms seems to be taking hold, which only added fuel to my desire to revisit those times. Then along comes the "Spectrum Next" project on Kickstarter. Unfortunately, I missed the first round, but definitely got in on the second. In anticipation of it's

arrival, I determined I was going to be making some games like the good old days. I bought an Agon Light, a cheap Z80 based open source system that had gained some popularity just about the right time, and set to remembering how to program in Z80 again. While this was fun, I soon realised that this was not what I wanted to do, I guess I'd become too spoiled by modern tools and technologies. So, I decided the best way to do what I wanted to do was to make a tool to do it. I'd already gained some experience with this, being responsible for the 3D Construction Kit at Incentive, and having worked on in-house tools at various games companies in my career, I knew this was something I could do and enjoy doing. And so Kwyll was born, as with all the best projects, a means of scratching my own itch, that garnered some interest from the community and grew into something else.

Getting Started

Installation

Kwyll is distributed as a simple archive of the appropriate format for each platform, a .dmg for Apple macOS, and a .zip for Windows and Linux. The main executable is contained within the archive. It is entirely self-contained, no installation is required, simply run the executable, and copy it to your preferred location for applications.

Export Toolchain

As of v0.1.0.9, Kwyll no longer requires the installation of a build toolchain, this section is left for anyone still using an earlier version, but the recommendation is to update to the latest version for faster export, no dependencies, and more available memory.

The only additional requirement is the build toolchain, this is delivered separately, it is based on a custom build of the z88dk compiler and is licensed under the same open source license as the z88dk project. Kwyll can download and install the appropriate version of the toolchain for you. The version downloaded is significantly cut down from the installation of the z88dk project itself, including only what is necessary to build a Kwyll game. To install the toolchain, open the 'Export' dialog by selecting the export button in the toolbar, or the 'Export-→Export Game' menu option. The first time you do this, the dialog will indicate that no toolchain is found, as shown in Figure 1.

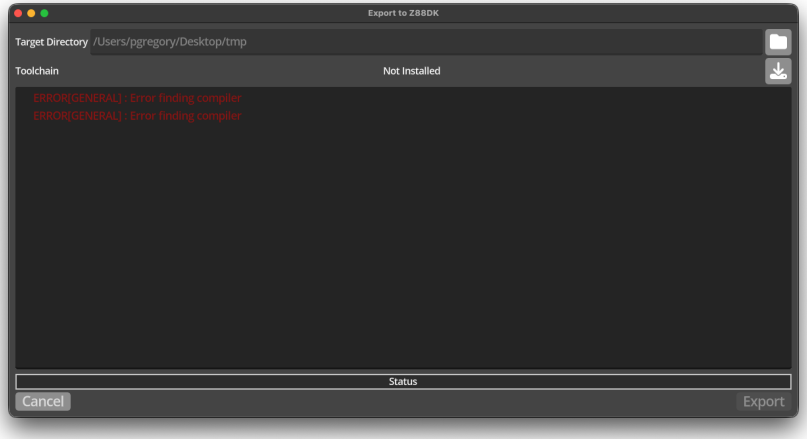


Figure 1. Export Dialog No Toolchain

If you select the download button to the right of the Toolchain row,  Kwyll will display the system it thinks you are running on.

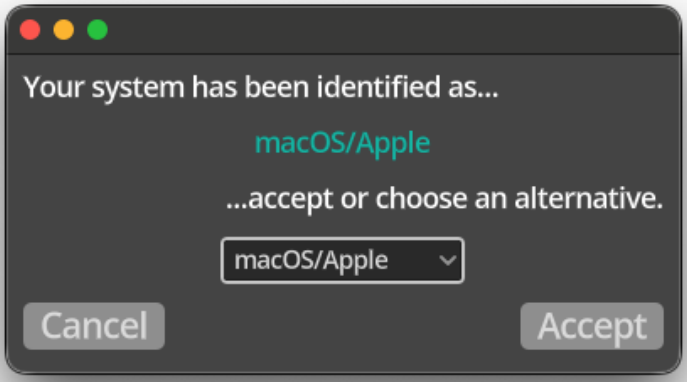


Figure 2. Toolchain Download

If the identification is correct, simply accept the choice, or alternatively select the correct system from the dropdown to override the identification.

Once accepted, Kwyll will download the correct toolchain and install it into the application folder for Kwyll, which differs by operating system.

macOS: `~/Library/Application Support/Godot/app_userdata/Kwyll`

Windows: `%APPDATA%\Godot\app_userdata\Godot\Kwyll`

Linux: `~/.local/share/godot/app_userdata/Kwyll`

Once installed, the export dialog should change to show the succcessful installation of the toolchain, and allow you to export your Kwyll game.

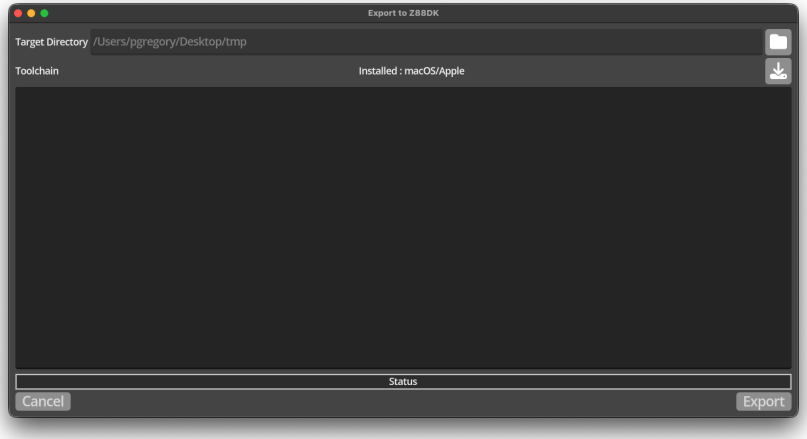


Figure 3. Export

Introduction

Terminology

There are several stages to developing a game with Kwyll, each with their own section in the tool. Below is a short introduction to some of the terminology that you will come across frequently throughout this documentation, this is a good place to start to ensure that you have the best chance of understanding what is being described later in the guide.

Sprites

Of course, any game needs some graphics, right? In Kwyll there are two main places you can draw pixel graphics, Sprites, and Tiles. A Sprite in Kwyll is a rectangular grid of pixels, Sprites can be various sizes, always in increments of 8 pixels in both axes. So, for example, 8x8, 16x16, 8x16, etc. An arbitrary limit is placed on the maximum size of 32 in either axis just to keep things within reason for the target platforms to maintain performance and limit memory use. See sprites for more details.

Tiles

Tiles are the simpler, more constrained, cousin to Sprites. They are also simple grid of pixels, however, a tile is always a fixed size (8x8 pixels on the Spectrum), and cannot be used in object types, instead they are used in room types, typically to draw the background of a room, over which an object instance sprites are drawn, and screens, for example to draw borders or background graphics for a screen. Tiles can also as the used in an instrument of the right type to show graphics instead of simple text or numbers.

Tilemap

In the Room Editor and the Screen Editor you can use **Tiles** to draw what is referred to in Kwyll as a **Tilemap**. This is an efficient way to draw lots of graphics in Kwyll. For room types, this is typically used to draw the background of your game. The screen tilemap is often used to

create a surround for your game window, something in which to place the instruments.

Brushes

Brushes allow you to optimise your background designs, specifically where there are sections of your backgrounds that are used frequently. Using brushes you can store a rectangle of tile information and repeatedly apply it to your tilemap very cheaply to maximise the available space for your game. See brushes for more details.

Object Types

An **Object** in Kwyll is a type of thing that can be displayed on screen and respond in various ways to the input of the player or to game logic. Objects are the heart of any Kwyll game, without objects, the game wouldn't actually do anything, making for a very dull game. See Object Types for more details.

Object Instances

An **Object Type** does not contribute to the game iself, it has to be used. **Object Instances** can be created in rooms and the map, each instance shares the visual representation and logic code from the **Object Type**, but each use has its own position, values for the variables defined in the logic, timeline animation, flags, and any other information that might be specific to a particular platform. See Object Instances for more details.

Room Types

A Room in Kwyll is the definition of a space in which part, or all, of your game takes place. Much like object types, they do not exist in the game on their own, they are used as locations on the map.

Timeline Animation

Objects added to room types can be animated using a simple keyframe system. Keyframes can be placed at certain times on a timeline that define the position of that object at that point in time. Kwyll will interpolate smoothly betwen the keyframes over time, allowing you to create smooth animations that are very cost effective in terms of

memory use. This is useful for enemies that follow a path for instance. See Timeline Animation for more details.

Locations

Just as with **Object Types**, a **Room Type** needs to be used in your game for it to have any value. A **Location** is how a **Room Type** is used on the map. A **Location** is equivalent to an **Object Instance**, each location shares the tilemap, logic, markers and room objects with any other location that uses the same **Room Type**. However, each location has its own name, default colour information, and variable values. See Locations for more details.

Map

The **Map** is where you place **Room Types** and arrange them to create your game's map. See the Map Editor for more information about creating your game map.

Screens

A **Screen** in Kwyll displays a collection of items including instruments, a tilemap, and the game window. **Screens** can be used for different purposes in your game, such as a menu, help screen, the game screen and others. See Screens for more details.

Instruments

Instruments are display elements that can be used in [Screens](#screens) to display various information about your game. They are used to present information to the player about the game, such as score, lives etc. See Instruments for more details.

Logic

In a Kwyll game, much of the way a game plays is controlled by the game **Logic**. Kwyll provides a node based system for creating logic that is easy to use and provides a more visual approach to programming than typical text based programming languages. See Logic for more details.

Variables

Variables are values that can be registered anywhere logic is used to store data during the game. When something that has logic associated with it is created, such as a location or object instance you can define the initial value of these variables to customise each instance of that type, such as defining its direction, or how much damage it can impart when attacking the player. See Variables for more details.

Data

In Kwyll, the **Data** section allows the game designer to define a fixed set of text strings to use in the game. These are typically used to display in instruments using the **Text** instrument type.

Input/Output

Player input in Kwyll is entirely managed by the logic system, the game designer is responsible for checking the input using the relevant nodes and responding to that input using the wealth of other logic nodes available. Input is handled in one of two ways, via the controller node, or the key input node.

Controller

The Controller Node can read input from either the defined control keys, which can be customised in the project settings and updated at runtime using the configure controller node, allowing your player to redifine their preferred keys for left, right, up, down and fire.

Key Input

The Key Input Node can be used to check for specific keys additional to those provided by the controller node. It can also be used to check for *any* key, if all that matters is the player presses a key to start the game for example.

Keycodes

In Kwyll keys are identified using an internal keycode system that can be adapted for use on various different target platforms. See Keycodes for more details.

Colour

TBD

Simulator

TBD

Coordinate System

In Kwyll there are several different coordinate systems used for different purposes. Understanding which coordinate system is being used is important when creating your logic to ensure that you are using the correct values for positions, movement, and other calculations. See Coordinate Systems for more details.

Backgrounds

Tiles

Tilemaps

In the Room Editor and the Screen Editor you can use Tiles to draw what is referred to in Kwyll as a Tilemap. This is an efficient way to draw lots of graphics in Kwyll. For Rooms Types, this is typically used to draw the background of your game. The Screen tilemap is often used to create a surround for your game window, something in which to place the instruments.

Both the Room and Screen tilemaps store a tile type and colour information for each grid cell in the tile map. The grid size is defined by the *Window* game setting for room tilemaps, and is the entire size of the screen for screen tilemaps. It should be noted however, that in a Screen that is defined as a Game Screen, and shows the game window, it is not possible to draw tiles in the area occupied by the game screen, they will be overridden by the game display.

Tilemap Collision Data

Room tilemaps also have an additional feature stored in their tilemaps that is specific to gameplay and not appropriate for Screens, collision data. This stores information about how objects can or cannot pass through a particular grid cell. It's possible to define each grid cell to allow or prevent movement through it in all 4 directions independently. For example, a grid cell can be configured to allow an object to pass through safely when travelling up the screen, but to prevent it passing through when travelling down the screen, something often used in platformer games for the platforms.

Brushes

In addition to drawing tiles, attributes (colours) and collision information directly into the tilemap, another tool available in Kwyll for drawing the content of tilemaps is the *Brush*. A brush consists of a separate record of tile, attribute and collision information stored within the game that can be placed any number of times into a tilemap very cheaply. See the Tilemap Editor section for details of how to create and use Brushes. Brushes are a very efficient way to create tilemaps that have lots of repeated sections. Drawing each individual tile is more expensive than storing the repeated sections in a Brush and just pasting the Brush multiple times. A Brush can be any width or height in tiles, not pixels.

Objects

Object Types

In order to use objects in your Kwyll game, you must first describe to Kwyll what your objects are and how they work, this is done by creating Object Types.

The type of an object includes things including what it looks like, via Sprites and any other information that is required to display the object that may be platform specific, such as colour and draw mode, and any Logic.

An **Object Type** consists of two main elements.

- Animations, which are sequences of Sprite images played in order to create the illusion of animation. Each object can have many Animations, for example, a player object might have an animation for walking left, one for walking right, one for jumping, one for attacking etc. The game designer can choose which Animation is played using the logic nodes based on what is happening in the game at that moment.
- Logic is the "brains" of an object, it defines what an object does, and when. In Kwyll, logic is created using a visual programming tool, a powerful means of writing code without writing reams of text.

Sprites

Of course, any game needs some graphics, right? In Kwyll there are two main places you can draw pixel graphics, **Sprites**, and **Tiles**. A sprite in Kwyll is a rectangular grid of pixels, sprites can be various sizes, always in increments of 8 pixels in both axes. So, for example, 8x8, 16x16, 8x16, etc. An arbitrary limit is placed on the maximum size of 32 in either axis just to keep things within reason for the target platforms to maintain performance and limit memory use.

It is important to know that a sprite doesn't actually 'do' anything in Kwyll, it's just an image, it has no position, it isn't by default drawn to the screen anywhere, in order to get a Sprite onto the screen, it must be used by an Object.

Draw Modes

On the Spectrum, when a sprite is assigned to an oject, you can choose which mode to use to draw the sprite to the screen, which is best depends on the particular use case in your game. The optons are:

LOAD

The sprite is just drawn into each 8x8 cell that it touches, completely overwriting anything that might already be there in the Tilemap or any other objects. This is normally only used if you know there will be no background where the object is used. This is the fastes sprite rendering mode.

OR/XOR

These two modes combine the pixels of the sprite with those in the background using bitwise OR and XOR operations. The result is that the background isn't completely wiped out in 8x8 blocks like LOAD mode, but depending on the nature of the sprite and the tiled background, can result in objects becoming difficult to see. This is suitable for cases where you know the background isn't too complex or detailed and you need speed. This is the second fastest sprite rendering mode.

MASK

This requires that a sprite is defined with a **Mask** layer, see the Sprite Editor for details of creating mask layers. This mode clears a part of the background in a user defined shape before drawing the sprite pixels into that space. It is the most visually effective draw mode, resulting in well defined pixels that can be easily distinguished from the background, but is the most costly in terms of both memory use, requiring additional data for each sprite, and in terms of performance.

Object Instances

An **Object Type** does not contribute to the game iself, it just defines what an object looks like and what it can do. Think of it as a template or blueprint for a particular type of object. The blueprint isn't actually an object, it's a description of certain properties that actual objects of this type will share in common. As an analogy, think of a car, the blueprint for the car is the **Object Type**, but it's not actually a car. When a car is built from the blueprint, that is the **Object Instance**, it will have certain properties common to all cars, 4 wheels, steering, an engine, etc., these are the properties defined by the **Object Type**. Each manufactured car, however, can have a different colour, a different interior material, different wheels, different trim, these are the properties that are stored with each instance to "specialise" that particular car. In Kwyll, the unique properties for each instance include a position, colour information, and values for the variables described in the **Object Type**. Object instances can be created in Rooms and on the Map.

Changing a variable value in object logic changes the value on the particular instance of that object that is running the logic. For example, imagine you have an **Object Type** "Enemy" with some logic that includes a variable "health" and in the logic for that **Object Type** you reduce the health variable value each time it is hit by a missile from the player. If you create two instances of that **Object Type**, each instance will have its own value for the "health" variable, when a missile hits the first enemy object, and the shared logic executes and reduces the "health" of the enemy, it is modifying only the value stored on the first instance, the value on the second enemy is not changed. Similarly, when a missile hits the second enemy, its health value is reduced, and the health of the first enemy is not changed.

Objects can be used in the following ways.

- Map Objects - these are created in the Map Editor and are constant throughout the game, they will always exist independent of which Location is currently shown, and will be drawn, if their position puts them on screen, and their logic will be run, if they are flagged as

"Active".

- Room Objects - these are in each Room Type, created in the Room Editor, and will only be visible and active while in a Location that is an instance of that **Room Type**. It's worth noting that when exiting a room that contains **Room Objects**, the data used by the Kwyll library for those objects, such as memory for a Sprite etc. will be freed, reducing the overhead for both memory and performance, so it is advisable to use **Room Objects** where appropriate over **Map Objects**.

It is important to be aware that Room Objects are unique to a **Room Type** not a location. So, if you move or otherwise change a **Room Object** while in one particular location, when entering another location that is also an instance of that same **Room Type**, the changes will affect that location too. As such, when using **Room Objects**, it is a common pattern to have a "Room Entered" logic trigger on the **Room Type** that configures the **Room Objects** on entry if you want some specific behaviour per location.

- Dynamic Objects - these are not created in an editor during your game creation, they are instead created during the game using the Spawn Object node, and can be destroyed using the Kill Object node. They are in all other respects similar to **Map Objects**, they are not constrained to a particular Room Type, and will continue to exist across changes in Location until they are removed.

Object Properties

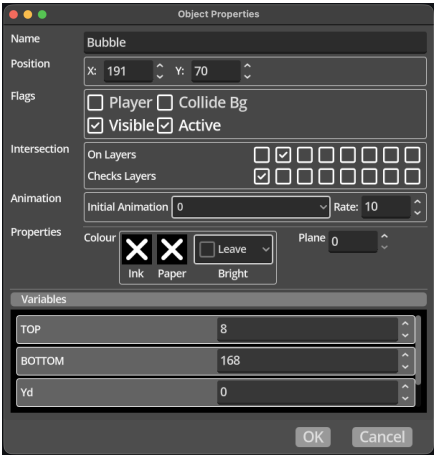


Figure 4. Object Properties Dialog

Each object instance has a number of properties that influence how the object works in the game. These properties are unique to each object instance, unlike the shared properties that all instances of a particuler Object Type have such as sprite animations and logic.

- Name - a meaningful name for this particular object instance.
- Position - the position of the object as an X and Y coordinate. Where this places the object depends on the type of object instance. A Global or Dynamic object will have coordinates that represent a position on the Map, while Room objects will have coordinates that represent a position within the room, irrespective of which location and where that location is on the Map. This must be considered when using logic to change the position of objects, and to check things like which tile is at a location related to an object.
- Flags - used to define the behaviour of the object.
    - Player - if the player flag is checked, the object will check if it is off screen as it is moved, and if so, whether to move to a new location on the Map. This is useful to make it simple to implement a game where rooms connect perfectly on the Map, and moving out of one location should naturally result in moving to the neighbouring location, without having to manually switch location in logic.

- Collide Bg - if this flag is checked, the object will be checked against the collision information assigned to each tile in the tile map. If a tile's collision information indicates an object cannot pass through a side, the object will be prevented from doing so.
- Visible - if this flag is checked, the object's sprite will be drawn if it is on screen. If it is cleared, it will not be drawn, but it will still interact with other objects, and its logic will still operate.
- Active - if this flag is checked, the object will be considered when checking if objects intersect, and whether the Object Hit trigger is fired when they do.

- Intersection - Kwyll can check if objects are intersecting each other and trigger logic when they do. In order to limit how much work is done to do this checking, Kwyll implements a layering system for objects. Careful use of this is imperative to limit the performance impact of intersection testing. The layering system works by marking each object as "on" one of the 8 possible layers, and defining which other layers it should check for intersections with. When checking a pair of objects, A & B, for intersection, Kwyll will only consider pairs where either object A is on a layer that object B is checking for, or object B is on a layer that object A is checking for. This gives a great deal of flexibility in limiting which things to check. For example, enemies can be be put on their own layer, and that layer is not included in their "checks" layer list, which would mean Kwyll will never bother checking for intersections between two enemies.

Use the "Performance" tab of in the Simulator, to monitor how many intersection pairs are being checked, a good way to improve the speed of your game is to reduce that number as much as possible with careful use of the layers.

  - On Layers - this defines which layer(s) the object is considered to exist on. An object can be on multiple layers, for example, you may decide that layer 2 is for collectibles, and layer 3 is for things that hurt enemies. An object can be placed on both, meaning the player can collect it, and if an enemy object touches it, it can be hurt.

    - Checks Layers - this defines which other layers to check for intersection with. When checking if a particular object is intersecting, only objects that are on one of the layers defined here will be considered.

- Animation - these properties control the sprite animation of the object. If an Object Type has sprite animations configured, you can choose which animation is playing by default, and at what rate. The rate is defined as the number of game frames between changes in a sprite frame. So, if the game is running at 50 frames per second, and the rate is set at 25, it will animate at 2 frames per second, waiting 25 game frames before updating the sprite frame.
- Properties - this section contains properties that are specific to each platform, such as colour and plane for the Spectrum.
    - Colour - allows you to define the colour of an object in terms of ink, paper and brightness. Each property can be specified, or left to the default. The method of identifying the default colour is described in Colour Priority.
    - Plane - When a sprite is drawn to the screen it is possible that there will be other sprites in the same place on the screen, so there has to be a way to decide which one is drawn first. This is the role of planes. Think of planes as "layers", with the smallest number being closest to the screen, and increasing numbers going further away. So an object that has its plane set to 1, will have its sprite drawn over the top of another object that has its plane set to 2, which in turn will drawn over another object that has its plane set to 5. This feature may be platform specific, check the details for your platform for details.
- Variables - this area a will show a list of the variable names that are defined on the Object Type for this object instance. It allows you to set the values of those variables at the start of the game. This is particularly useful to enable configuration of objects that share a common type, but have specific behaviour per instance. For example, an enemy object might have logic that makes it patrol horizontally or vertically between two points in the room. Defining variables to indicate which direction and the minimum and maximum position in

that axis means you can easily set those per instance to create variations of an enemy type very efficiently.

Locations

Room Types

A game in Kwyll consists of a number of locations on the Map, each location represents a separate space in the game, Kwyll does not currently support scrolling so gameplay is constrained to a single location at a time. Each location is based on a **Room Type**, which is in many ways equivalent to an Object Type.

The **Room Type** is responsible for defining shared properties among all Locations that use that **Room Type**, much like an Object Type defines common properties that all Object Instances share. These properties include:

- A Tilemap, which represents the background of the room, such as the scenery, walls, platforms, etc.
- Room Objects, which are unique to that room type and are only active while an instance the room is the current location.
- Room Logic that is specific to the room type. Logic triggers on a Room only run when a Location that uses the Room is the current game location.
- Markers, very lightweight objects that only contain position information, no sprites or logic, and are used primarily within Room or Object Logic to place things within a room dynamically, for example, a marker may be used as a spawn point for the player, or as a position to appear when entering a room from another room.

A **Room Type**, much like an **Object Type** doesn't actually contribute to the game itself until it is used. A **Room Type** is used to define one or more Locations on the Map. A single **Room Type** can be used multiple times on the Map. For example, it might be useful to create a room that has some logic on the Room Entered trigger to alter how the room looks and works using Brushes or other logic nodes based on variables defined in the **Room Type** and set in the location on the Map. This way you can place a single

Room Type on the Map multiple times, and change how it looks by editing the Variables for that specific Location.

Room Types that are not placed in the Map at all, will not be exported and will therefore not contribute to the total memory use of your game. This can be useful to allow you to use a **Room Type** as a "scratch" area, for experimenting with tiles and brushes without worrying about using up valuable space.

Locations

Just as with **Object Types**, a **Room Type** does not contribute to the game itself, it just defines what a room looks like and can do. A location is the equivalent of an Object Instance, it is the instance of a **Room Type** in the game. Locations are created by dropping room types onto the Map in the Map Editor.

Changing a variable value in room logic changes the value on the particular location that is running the logic. For example, imagine you have a forest **Room Type** with some logic that includes a variable "south door", which is 1 if the south door is closed, and 0 if it is open, and in the logic for that **Room Type** you paste a brush in the south doorway, an open door if it is open, a closed door if it is closed. If you create two locations using that **Room Type**, each location will have its own value for the "south door" variable, when something happens to open the door in the first of those two locations, the value of the "south door" variable is updated, and the shared logic changes the display of the door, it is modifying only the value stored on the first location, the value on the second location is not changed. Similarly, when the player opens the south door in the second location, the "south door" variable on the second location is changed, and the first location's "south door" value remains unchanged.

Location Properties

Figure 5. Location Properties Dialog

Each location has a number of properties that influence how the location works in the game. These properties are unique to each location, unlike the shared properties that all instances of a particuler Room Type have such as tile map and logic.

- Name - a meaningful name for the specific location on the map.
- Background Colour - allows you to define the default background of a location in terms of ink, paper and brightness. Each property can be specified, or left to the default. The method of identifying the default colour is described in Colour Priority.
- Variables - this area a will show a list of the variable names that are defined on the Room Type for this location. It allows you to set the values of those variables at the start of the game. This is particularly useful to enable configuration of locations that share a common room type, but have specific behaviour per instance. A good example of using this facility of Kwyll is to define one room type, add a number of variables to the logic to choose the layout of the room, such as which sides have doors, what type of walls to use, etc. Then, in the room logic, use those variable values to change the appearance of the room. This

way, you can place multiple instances of the same room type on the map, tweak the variables in the location properties and get many different locations from a single room type. The advantage of this is each location is relatively lightweight in terms of memory use, requiring only the unique location properties such as variable values, allowing you to define a larger map very efficiently.

Map

The **Map** is where you place **Room Types** and arrange them to create your game's map. In the Map Editor, you can drag **Room Types** from the asset list to drop them into place on the map, this creates a Location on the **Map**. How you arrange your locations on the **Map** depends on the type of game you're creating. In some games, the arrangement of locations on the **Map** is not important, the Kwyll logic can be used to move from location to location depending on other things, such as interaction with an object like a doorway. In other games, if a player exits the game window in one of the 4 possible directions, it is expected that they change to the location in that direction on the **Map**. This is facilitated by the Player flag on object instances. If an object has the **Player** flag set, when it exits the game window for any reason, Kwyll will check automatically if there is another location in that direction and move the game to that location. In this example, it's important that the locations are placed accurately so that they connect.

It's important to note that, while the locations must be connected carefully they do not have to perfectly align into a grid. As long as there is a location on the map at the place where the player exits one location and enters another it will work, for example, a staggered alignment as shown in Figure 6, will work fine.

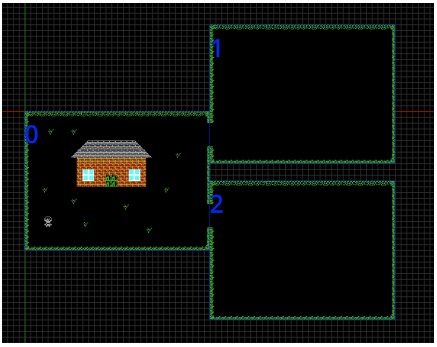


Figure 6. A Staggered Map

Timeline Animation

Figure 7. The Timeline Editor

When **Object Types** are added to a **Room Type** as an **Object Instance**, it is possible to animate the position of those instances in the room without using logic for simple animation requirements, such as following a path. In the Room Editor there is an animation timeline editor which represents time in frames. With this editor you can place "keyframes" at any point in time (up to 65535) which hold a position for the object being animated. At runtime, the Kwyll engine will interpolate between those keyframes over time. So you can place a keyframe at time 0 and one at time 50, and the position will change between the position stored on the first and the position stored on the second over 50 frames.

Timeline animation also provides another useful feature, it is possible to trigger logic at any point in the timeline, so you can have an animation that moves an object along a path, and at certain points in that path triggers logic on the object to perform a particular action, such as changing the sprite animation, or firing a projectile.

Screens

Screens

A **Screen** in Kwyll is a collection of Instruments, a Tilemap, and optionally, the game window. A **Screen** can also have logic associated with it that is run while it is the current screen.

A **Screen** can optionally be defined as a "game screen", which means it will display the game window, the game window can be placed arbitrarily within the **Screen**, and different screens can have the game window in a different place, the game window size however, is fixed for all screens.

There is only one **Screen** active at any time in a Kwyll game. Uses of screens include the a menu screen, instructions, and the main game screen which will have the game window positioned on the screen accordingly.

Instruments

Instruments are display elements that can be used in Screens to display various information about your game. An instrument can be one of 4 types:

- Integer, a simple numerical value.
- Text, a fixed text string from the list of options defined in the Data section.
- Tile, the graphic for a specific tile from the tile set defined in the Screen Editor or Room Editor.
- Keycode, displays the key name for a particular keyboard scancode.

Each instrument has a position in the same grid as the tile map, that is, it is constrained to 8x8 pixel positions, you cannot place an instrument with pixel accuracy. Each instrument can also optionally have its own colour information, if not provided, the instrument will take on the colour settings

from the Screen.

The **value** of an instrument is always a 16 bit numerical value. How this value is interpreted to produce the display depends on the type of instrument:

- Integer, the value is displayed in base 10.
- Text, the value is the index of a text string in the strings list defined in the Data section.
- Tile, the value is the index of a tile in the tile set, you can determin what each tile's index is by looking at the tile set in the Screen Editor or Room Editor, the white (leftmost) number below the tile image is the index.
- Keycode, the value is converted from a key Keycode to a key name as text, for example, 1792 will display "SPACE".

You can modify the value of an instrument during the game using the Set Instrument node.

Color

Colour Priority

Logic

Logic

In a Kwyll game, much of the way a game plays is controlled by the game **Logic**. Kwyll provides a node based system for creating logic that is easy to use and provides a more visual approach to programming than typical text based programming languages. It is described in more detail in the What is Logic in Kwyll? section of this manual.

Logic can be used in various places in a kwyll game allowing lots of flexibility in how you structure your game.

Object Logic

Each Object Type can have **Logic** assigned to it, this logic is shared with each **Object Instance** that uses the definition, allowing you to very efficiently define behaviour that is shared among many objects. Object logic only runs for objects that are currently active, that includes any Map level objects which are global and active all the time, any dynamic objects, and any room objects for the current location only.

Room Logic

Each Room Type can have **Logic** assigned to it in the same way as an Object Type. Each Location that uses the room definition shares the logic in the same way an Object Instance does. Room logic only runs for the current Location.

Screen Logic

Each Screen can have Logic assigned to it, this is typically used to manage the elements in the screen, primarily the instruments. Screen logic only runs for the current screen.

Game Logic

In addition to each of the individual locations listed above, a Kwyll game can have some global logic that will execute all the time irrespective of location and screen.

Shared Logic

Each of the logic locations listed above are potentially active logic sections, that means that Kwyll will run certain flows on them at the appropriate time, for example the Always flows will run ever frame, the Object Hit flows will run when an object intersection is detected. Shared logic is different. It offers a means to create sections of logic that can be reused, effectively creating custom nodes. Shared logic consists of **Subgraphs** which are equivalent to subroutines in many common programming languages. They can take an arbitrary number of inputs and output an arbitrary number of results.

Subgraphs are a very powerful concept in Kwyll that gives both the ability to optimise and streamline your logic creation with custom nodes, and share those custom nodes with others.

Variables

TBD

I/O

Keycodes

Keyboard input in the [Key Input](../../logic/nodes/key_input.md) and the [Configure Controller](../../logic/nodes/configure_controller.md) nodes use a custom code to reference keys that ensures that Kwyll keycodes are not platform specific if at all possible. The scancodes are defined using a "section" and "index" mapping, that is similar to the Spectrum method of mapping key presses to input codes, but is flexible enough that the same mechanism can be used on other platforms. Each keycode is a 16 bit integer with the high byte being the section number, and the low byte being the index in that section. This is flexible enough to accommodate a single section with 256 keys, or multiple sections with fewer keys.

The Kwyll keycodes for the Spectrum are split into 8 sections of 5 keys:

Table 1. Keycode Table

	0	**1**	**2**	**3**	**4**
0	CAPS	Z	X	C	V
1	A	S	D	F	G
2	Q	W	E	R	T
3	1	2	3	4	5
4	0	9	8	7	6
5	P	O	I	U	Y
6	ENTER	L	K	J	H
7	SPACE	SHIFT	M	N	B

So, for exmaple, the scancode for "SPACE" is calculated as:

Row 7 in the high byte, and index 0 in the low byte, which in hexadecimal is:

```
0x0700
```

or:

```
1792
```

in decimal.

Coordinate Systems

Introduction

There are several different coordinate systems used in Kwyll, each with its own purpose and conventions. Understanding these coordinate systems is crucial for correctly positioning objects, handling movement, and performing calculations within the game. This section provides an overview of the different coordinate systems used in Kwyll and how they relate to each other.

Global Coordinate System

The global coordinate system is used in the map editor for placement of locations and objects. It is a 2D coordinate system where the origin (0,0) is typically at the top-left corner of the Map. The x-axis extends to the right, and the y-axis extends downwards. This coordinate system is used for defining the layout of the game world and positioning elements within it. The origin of the global coordinate system is displayed in the Map Editor as the crossing of the red and green lines in the grid view.

In logic, you can use the Map to Room and Room to Map nodes to convert between the global coordinate system and the room coordinate system.

Figure 8 shows the Map Editor, the origin of the global coordinate space is shown highlighted in magenta, and the two cyan arrows show how a Location and a map level Object are located relative to the origin.

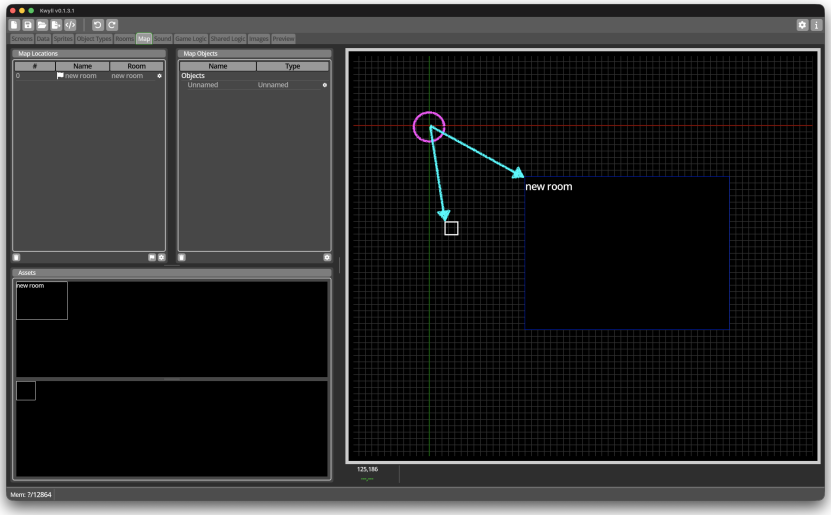


Figure 8. Map Coordinates

Room Coordinate System

The room coordinate system is used for positioning objects and markers within a specific room. It is also a 2D coordinate system, but the origin (0,0) is at the top-left corner of the Room. The x-axis extends to the right, and the y-axis extends downwards. This coordinate system is used for defining the layout of individual rooms and placing objects within them. The origin of the room coordinate system is displayed in the Room Editor as the crossing of the red and green lines in the grid view.

In logic, you can use the Map to Room and Room to Map nodes to convert between the global coordinate system and the room coordinate system.

Figure 9 shows the Room Editor, the origin of the room coordinate space is shown highlighted in magenta, and the cyan arrow shows how a room Object is located relative to the origin.

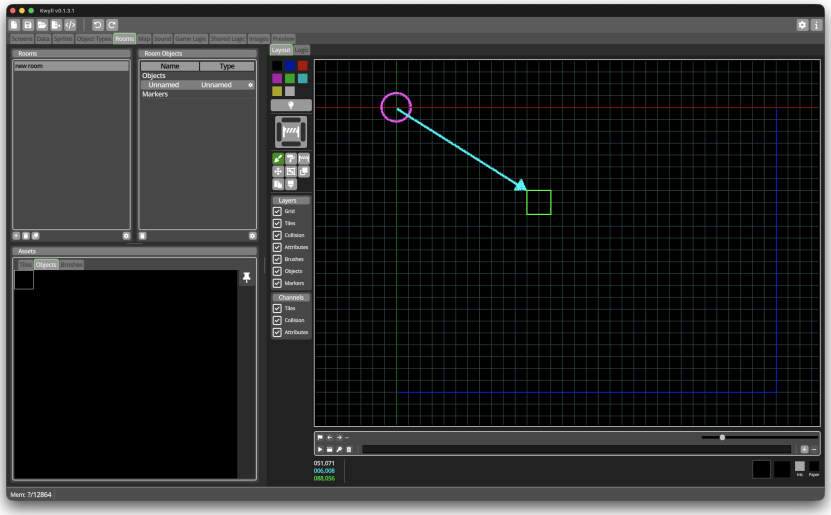


Figure 9. Room Coordinates

Tile Coordinate System

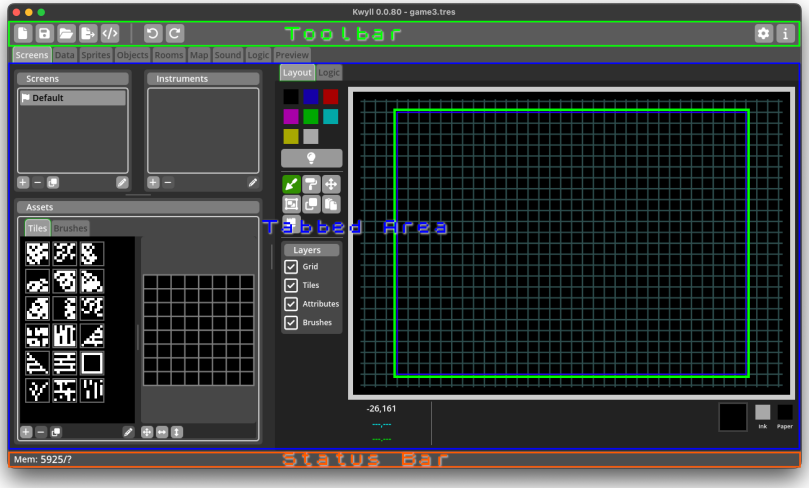
The tile coordinate system is for positioning tiles in a Tilemap. It is different to the global and room coordinate systems in that it is based on "cells" rather than pixels. Each cell corresponds to a tile in the tilemap, and the coordinates are defined in terms of these cells. The origin (0,0) is at the top-left corner of the tilemap, with the x-axis extending to the right and the y-axis extending downwards. This coordinate system is used for defining the layout of tilemaps and placing tiles within them.

Tilemaps are used in two places in Kwyll, in Screens, and in Rooms. The origin of the tile coordinate system is the origin of the tilemap, which is displayed in the Screen Editor or Room Editor as the crossing of the red and green lines in the grid view.

In logic, you can use the Pixel to Tile and Tile to Pixel nodes to convert between the pixel coordinates in either the global or room coordinate system and tile coordinates.

Interface

The Kwyll Interface



The main interface of Kwyll consists of a single window with a menu, toolbar, a tabbed main area and status bar. The menu is not shown in Main Interface, it is an operating system menu and as such on macOS displays at the top of the screen as per operating system guidelines, on Windows and Linux it may show at the top of the window above the toolbar.

Toolbar

The toolbar contains shortcuts to commonly used functionality provided by the menu functions, in order from left to right:

New Game

Clear the current game and start a new empty game.

 **Save Game**

Save the current game.

 **Load Game**

Load an existing game, replacing the currently open one.

 **Export Game**

Export the current game and build for running on a device or emulator.

 **Review Export**

Display a breakdown of the exported project for review, showing important information such as how much memory is used by different parts of the game.

 **Undo**

Undo the last action taken. Kwyll keeps a list of all actions taken up to the current point, allowing you to step back in history to undo changes made or redo them. Once any number of undo steps are taken and a new change is made the changes that could be redone are cleared and it no longer becomes possible to redo from that point.

 **Redo**

Redo the last action undone.

 **Settings**

Edit game settings.

Information

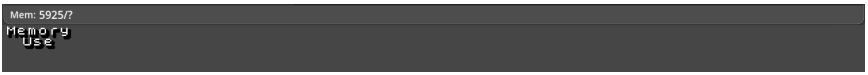
Show information about the version and status of your Kwyll installation.

Tabbed Area

The main body of the Kwyll window is taken up by the tabbed area. Each editor that Kwyll has is in a separate tab in this area, including Screen, Data, Sprites, Objects, Rooms, Map, Sound, Logic and Preview.

Status Bar

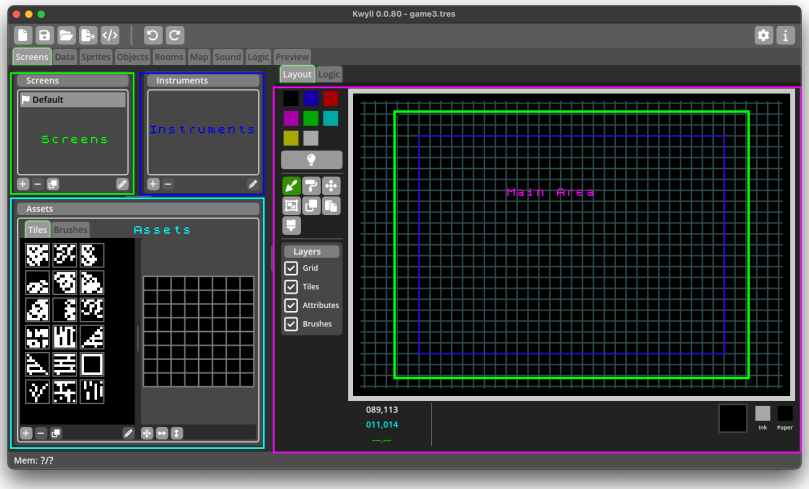
The status bar at the very bottom of the window is used to provide information about the game to the designer such as memory use.



Memory Use

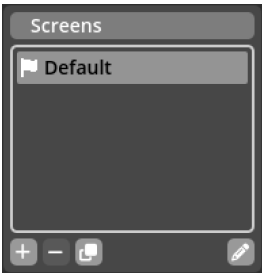
Shows how much memory, in bytes, the game data currently uses out of how much is available for use. If the usage exceeds the available space, the game will not work on device or emulator.

Screen Editor



The *Screen Editor* is where in Kwyll you get to create and modify Screens. It is organised into four main areas, the Screens List, the Instruments List, the Assets Panel and the Main Area.

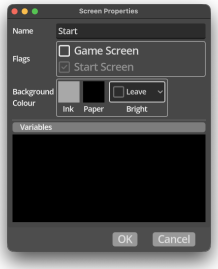
The Screens List



This is where you can select, add and remove the various screens that you have in your game. The toolbar below the list includes buttons on the left hand side to create a new screen, delete the currently selected screen, and duplicate the currently selected screen. And on the right hand side, it also

has a button to edit the screen properties.

Screen Properties



The *Screen Properties* dialog is shown when double clicking a screen in the screens list, or clicking the screen properties button. This is where you can modify various properties of the currently selected screen that are not editable in the main area layout and logic tabs.

Name

The name of the screen. This is used in logic for example when selecting to switch to a different screen, the Switch Screen node will list the screen names in a dropdown control to make it easy to select which screen to switch to.

Game Screen?

This toggles whether the screen should contain the game window or not. Typically only one screen in your game will have the game window included, but it is possible to have it in multiple screens, even in different places on the screen. Some screens, such as a menu screen, a help screen, etc. will not require the game window to be shown, they will consist entirely of instruments.

Start Screen?

This toggles if the screen should be the initial screen visible when starting your game. Only one screen can have this flag set, so selecting it here will automatically clear it on all other screens. The start screen

is also shown in the Screens List with a flag icon.

Ink, Paper and Bright

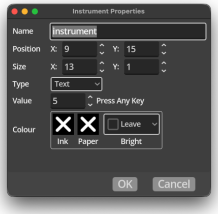
These combined set the default colours for the current screen. Instruments can override these values. The paper colour is used as the background to the entire screen. These values are also used as the default colours for tiles in the screen tilemap unless replaced by painting attributes in the tilemap, or by assigning colours to tiles.

Variables

In this section you can set the default values for any variables declared in the logic for this screen. Each screen can have its own logic, and in that logic you can declare a number of variables, they will be listed here for you to set the initial value that will be assigned to them when the game starts or restarts.

The Instruments List

This is where you can select, add and remove the various Instruments defined in the currently selected screen. The toolbar below the list includes buttons on the left hand side to create a new instrument, delete the currently selected instrument, and duplicate the selected instrument. And on the right hand side, it also has a button to edit the instrument properties.



The *Instrument Properties* dialog is shown when double clicking on an instrument or clicking the instrument properties button. This is where you can modify the various properties of an instrument.

Name

The name of the instrument. This is used in logic for example when selecting to update an instrument value, the Set Instrument node will list the instrument names in a dropdown control to make it easy to select which instrument to update.

Position

This is the position on the screen for the top left corner of the instrument. Instruments are placed on a grid of cells, each 8x8 pixels in size, so on the Spectrum, the range of these values are 0 to 31 in X and 0 to 23 in Y, any values beyond this range will result in the instrument being at least partially off screen.

Size

The size in cells of the instrument, as with the position, the instrument size is specified in terms of cells, each 8x8 pixels in size.

Type

The type of the instrument. An instrument will display its Value in different ways depending on the type of instrument. See Instruments for more information..

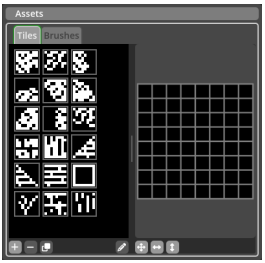
Value

The initial value of the instrument, the meaning of the value depends on the instrument type.

Color

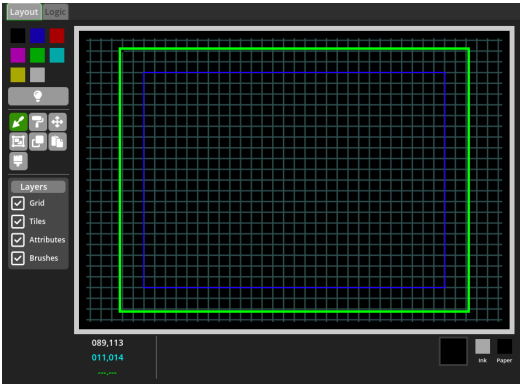
These controls allow you to override the default colour information defined by the screen. Ink and paper can be specified separately or ignored, choosing the black colour with an "X" indicates that the instrument should not change the default for ink or colour and instead use the value specified in the screen. The brightness value can on or off, only if one of ink or paper is specified.

Assets Panel



The *Assets Panel* provides access to assets that can be placed in the screen editor view in addition to Instruments. It is a common element used in more than one editor in Kwyll, and as such has its own separate section of the documentation here.

Main Area

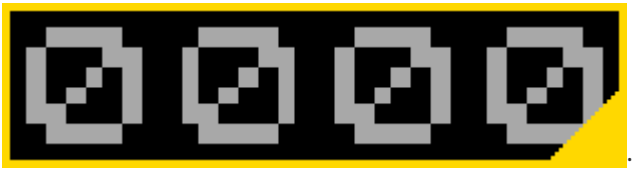


The rest of the *Screen Editor* is taken up with the main area, where the majority of the editing of a Screen takes place. It consists of two tabs, Layout and Logic. The Logic tab is a standard Logic Editor component that is used to edit logic that applies to the current room, see Logic for more details.

The Layout tab contains the screen layout editing tools. It consists of two parts, a standard Tilemap Editor, and layered on top of it a specific set of editing features for Screens.

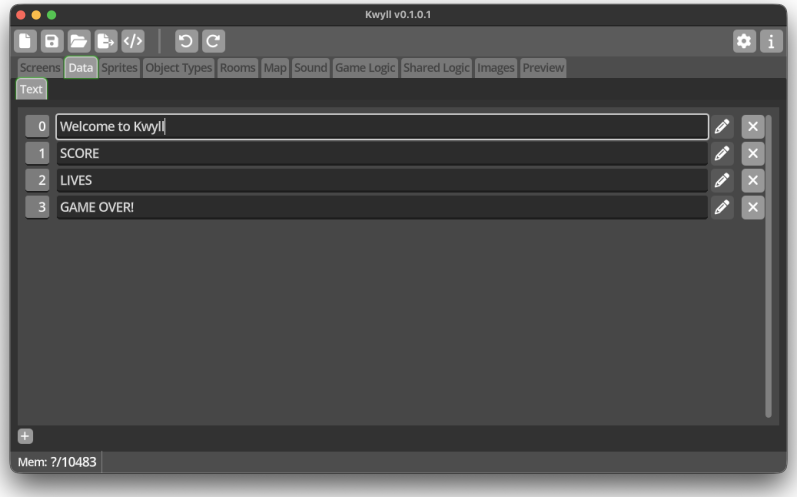
Layout

The screen *Layout Editor* works as a set of additional tools that operate in the same space as the *Tilemap Editor*. It provides visual tools for interactively placing screen elements such as Instruments and the *Game Window*.

An Instrument is represented on the grid as a yellow rectangle with a small triangle in the bottom right corner  . You can click and drag the instrument on the grid, it will automatically snap to the grid cells of 8x8 pixel increments. By carefully clicking on the small triangle in the corner, you can drag out the rectangle to change its size, again, it will automatically snap to the grid.

If the screen being edited has the *Game Screen* flag set in the properties, a blue rectangle will show to represent the *Game Window*. This element does not have a resize handle (triangle), as it's not possible to resize the game window in a screen, the game window size is fixed on all screens, the only way to modify the game window size is in the Settings dialog. However, the game window can be positioned independently on each screen it is used, so dragging the blue rectangle representing the game window works in much the same way as an *Instrument*, it will also snap its position to the grid.

Data Editor

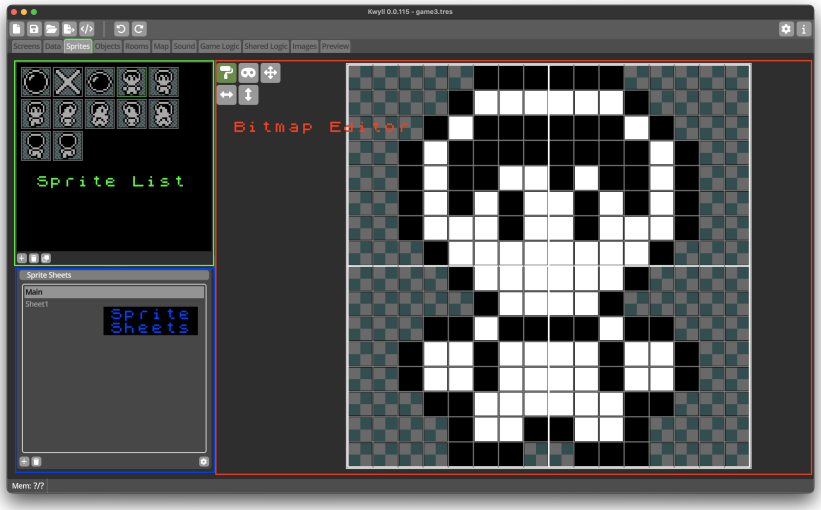


The *Data Editor* is where you can define text strings to be used in your game. The use of text strings is restricted to the Instruments on a screen. To the left of a each string in the list is a number, this is the number to assign to the *Value* of a text instrument to display this string, or to use in the **Set Instrument Value** node in logic.

The *Data Editor* is simple, in the toolbar below the list is a single button to create a new string. In the list, each string is represented by a row, with the first column being the index, used to display the string, the string contents follow, and a button to delete the string.

The edit button is currently not operational, reserved for future functionality.

Sprite Editor



The *Sprite Editor* is where you create your graphical masterpieces for use in your Kwyll game. It consists of three main areas, the Sprite List, the Sprite Sheets list, and the Bitmap Editor.

Sprite List

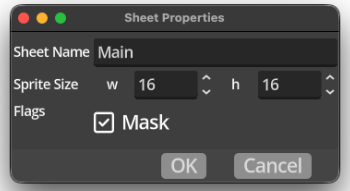
This section shows a thumbnail list of all the sprites in the currently selected sheet. It has buttons in the toolbar below to create a new sprite in the selected list, delete the currently selected sprite, and duplicate the currently selected sprite.

Sprite Sheets

This section lists the different sprite sheets defined in your game. Each sheet has a fixed size for the sprites in it, so if you need different sized sprites in your game, you'll need to create multiple sheets. The toolbar below the list has buttons on the left to create a new sheet, and delete the

currently selected sheet, as well as a button to edit the sprite sheet properties to the right.

Sprite Sheet Properties



The *Sprite Sheet Properties* dialog allows you to edit various properties of the selected sprite sheet.

Sheet Name

The name of the sprite sheet, give it a meaningful name to make your life easier.

Sprite Size

The width and height of the sprites in this sheet, in increments of 8 pixels in each direction, up to 32x32.

Flags

The "Mask" flag defines whether the sprites in this sprite sheet can have a mask defined. This is used when the sprite is to be drawn using the "MASK" mode. If the sprites in a particular sheet are only going to be drawn using "LOAD", "OR", or "XOR" mode, disable this flag to reduce wasted memory. See Sprites for more information about draw modes.

Bitmap Editor

This is where you draw your **Sprite** images. It has a simple set of tools in the toolbox area, and a pixel grid that is the correct size for the selected

Sprite.


The two main tools, the paint roller and the mask, define which type of pixels you draw or delete. If the paint roller is selected, the left button paints a pixel, the right deletes it. If the mask tools is selected, the left button draws a mask pixel, which is shown as a grey/teal checker pattern, and the right button deletes the mask pixel. The area in the bitmap which is displayed with the checker pattern is where the background pixels will show through.

The remaining tools are, in order, left to right, top to bottom:

Move


Selecting this tool and dragging on the bitmap will move the pixels and mask pixels, wrapping around when they go off an edge of the grid.

Mirror

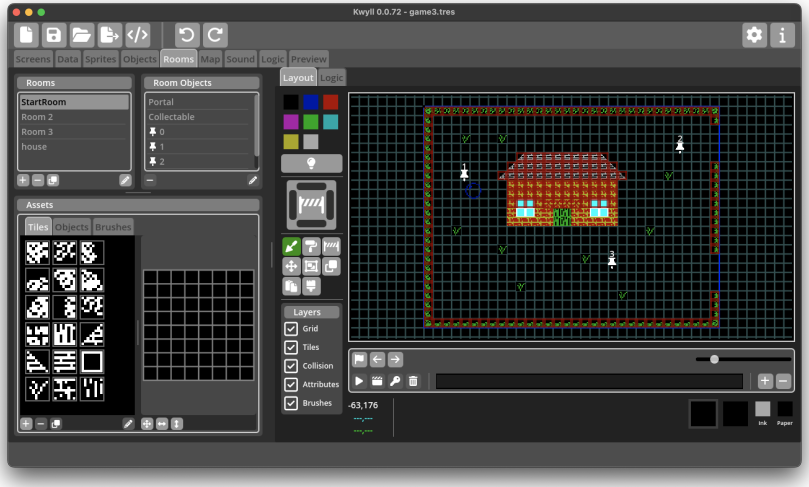

This is not a mode tool, it has an immediate effect when clicked, it will flip the pixels and mask of the **Sprite** in the horizontal direction.

Flip

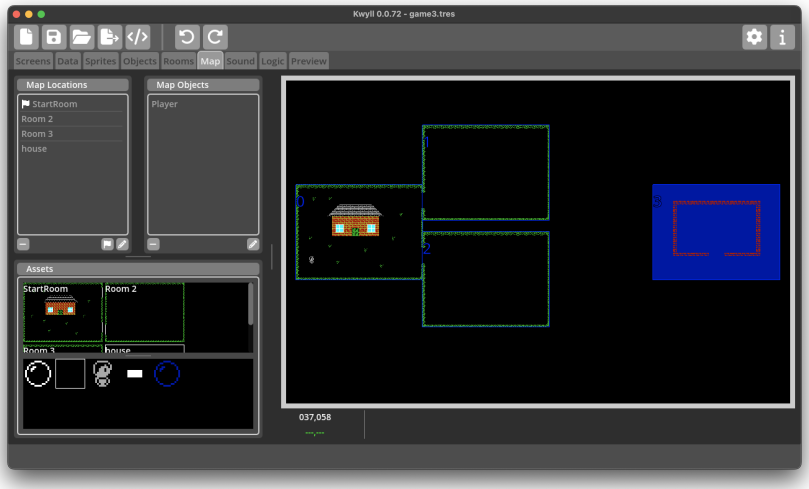
 This is also an immedate effect tool, when clicked it will flip the pixels and mask of the **Sprite** in the vertical direction.

Object Editor

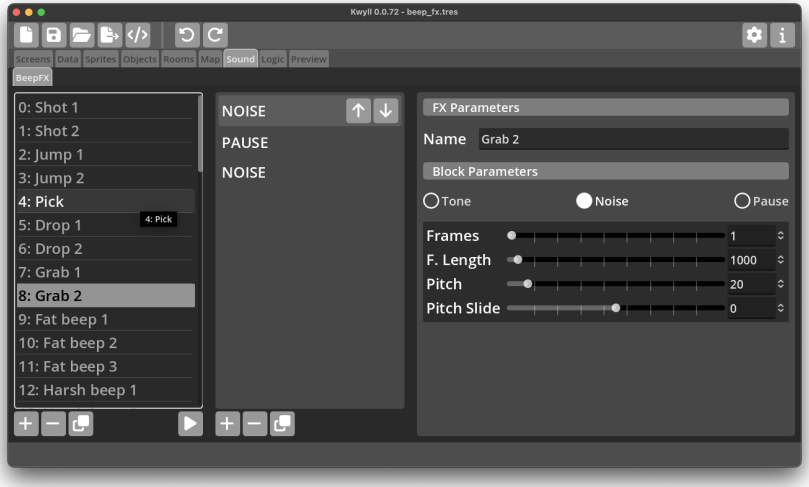
Room Editor


Kwyll 0.0.72 - game3.tres
Screens
Data
Sprites
Objects
Rooms
Map
Sound
Logic
Preview
Rooms
StartRoom
Room 2
Room 3
house
Room Objects
Portal
Collectable
0
1
2
Assets
Tiles
Objects
Brushes
Layout
Logic
Layers
Grid
Tiles
Collision
Attributes
Brushes
-63,176
Ink
Paper

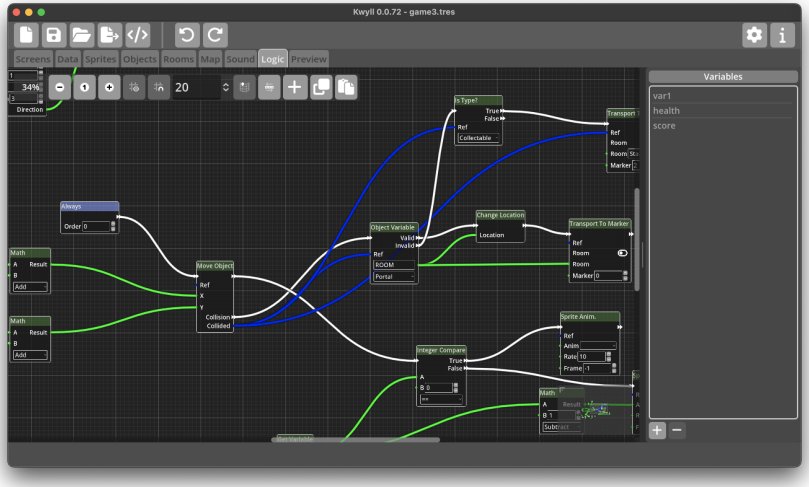
Map Editor


Kwyll 0.0.72 - game3.tres
Screens
Data
Sprites
Objects
Rooms
Map
Sound
Logic
Preview
Map Locations
StartRoom
Room 2
Room 3
house
Map Objects
Player
Assets
StartRoom
Room 2
Room 3
house
037,058

Sound Editor



Logic Editor

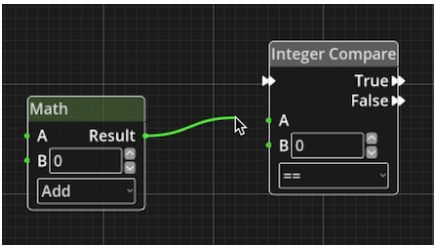


Editing Nodes

Nodes are created by right clicking the mouse in an empty area of the grid or pressing ++shift+a++, a pop-up menu will appear with a list of all the possible *Node* types as a tree organised by category. The popup has, in addition to the list of nodes, a text entry field that will be focused by default, typing in here will filter the nodes list to only those that contain the typed text, this is a very efficient way to find the exact node you're looking for. As you type in the search field, the list will instantly adapt to show only the matching nodes, and the first in the list will be selected. You can add the selected node by pressing return, or use the ++up++ and ++down++ arrow keys to move among the nodes that match the search term to select before pressing return. As you become familiar with the available nodes, this will be the most efficient way to add nodes to the logic graph. Initially it may be better to use the mouse to scroll through the list and click on the node you require directly. Below the list of nodes is a small text field that will change to show a short description of the currently selected node.

When a node is added to the graph, it will appear at the current mouse position.

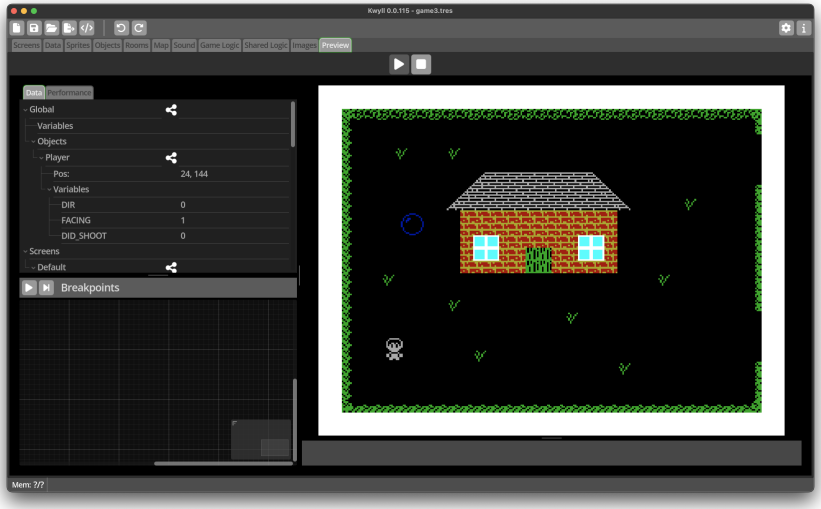
Click and drag on *Nodes* to move them around the graph and organise your program. You can ++ctrl++/++cmd++ click on multiple *Nodes* to select more than one at a time, or click and drag in space on the grid to drag a rectangle around a selection of *Nodes*.



To connect *Ports* left click on one of the two *Ports* and drag, a wire will appear connected to the *Port* you clicked on and the mouse. Drag the mouse to the other *Port* and when you get close enough, it will snap to the *Port*, release the mouse to complete the connection. If you struggle to begin the connection operation because the port is too small, it can sometimes be helpful to zoom in, this can be achieved using the mouse wheel or pinch gesture. *Ports* can be connected in either direction, it makes no difference to the way they work, data only flows from output to input.

To disconnect a port, click on the input end of the wire, the end going into the input port of a *Node*, drag it away from the port and it'll disconnect, you can reconnect to another port while still holding the mouse, or release the mouse with the wire disconnected to delete the wire altogether.

Preview


Kwyll 0.0.115 - game3.tres
Screens
Data
Sprites
Objects
Rooms
Map
Sound
Game Logic
Shared Logic
Images
Preview
Data
Performance
Global
Variables
Objects
Player
Pos:
24, 144
Variables
DIR
0
FACING
1
DID_SHOOT
0
Screens
Default
Breakpoints
Mem: ??

Assets Panel

Tilemap Editor

Settings

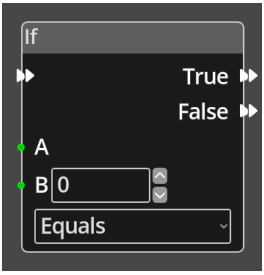
Logic

What is Logic in Kwyll?

One of the most important aspects of a game created with Kwyll is the ability to make objects, and other things, "do stuff", this is achieved with the logic system that is built into Kwyll. It is akin to the programming language in other game creation tools, but instead of being text based, it is visual, working by wiring together small items that do a relatively simple thing so that the whole does something more interesting.

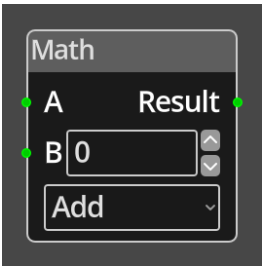
These "small items" are called *Nodes*, Kwyll provides a basic set of *Nodes* that can be combined to do a variety of things, the process of "wiring" them together enables each *Node* to either pass data to another node, or to control the flow of the "program". Some of the *Nodes* are very simple, such as the *Math Node*, whose only job is to take in two values, perform a basic mathematical operation on them, add, subtract, multiply or divide, and then output the result. Others are more complex and perform a detailed operation in and of themselves based on the input and parameters.

Each node has an optional set of input and output *Ports* which can be of different types depending on their purpose, and a set of *Parameters* which configure the operation of the *Node*.



This is a relatively simple *Node* that demonstrates most of the key features of a logic Node. Here you can see on the left of the *Node*; 3 input *Ports*, one with a double arrow at the top, and two green ones below. The double

arrow port is a *Flow* port, it is connected to other ports of the same type to control the flow of execution. A *Node* will only have at most 1 *Flow* input port. The green ports are data inputs, they provide integer numerical values to the node from some other node. You'll notice that the "B" *Port* also has a numerical input field alongside it, this is a parameter, in this particular case the value that the *Node* uses can either be provided by another node via the *Port* or specified as a fixed value in the parameter. In this example, you may want to compare the input from another *Node* at A with either an input from another *Node* or with a specific fixed value, 0 in this case. If you connect another *Node* to the input *Port* B, the numerical input field will be hidden as it is no longer necessary. On the right of the *Node* are the output ports, in this example there are two *Flow Ports*, True and False. This *Node* is a "branching" node, it will perform a comparison of the two values A and B using the specified comparison type, for example "Equals". If the comparison results in a true result, i.e. the two values are equal, the flow will follow whatever is connected to the True output *Port*, if anything. If the comparison is false, A and B are not equal, the flow will follow the False output *Port*. Below the B port is another *Parameter*, in this case it is the type of comparison to be made between A and B, equal, greater than, less than, etc. This *Parameter* has no input port, so it can only have a fixed value chosen at deisgn time. It is a means of configuring the operation of a *Node* in a way that doesn't require input from another *Node*. To summarise, on this *Node* A is a *Port* only, it requires an input wire to work, B is an input *Port* **or** *Parameter* you can choose how this value is set depeneding upon your needs, and the comparison type is a *Parameter*, you must choose this value when editing and it cannot change at runtime.



Some nodes will have no *Flow* ports at all, this means the *Node* is not meant to be part of the flow, but instead is meant to provide data to other

nodes that are part of the flow. These *Nodes* will have output data *Ports* that can be connected to the input data *Ports* of other *Nodes*. When a *Node* that is part of a flow has a wire connecting one of its input *Ports* to the output *Port* of another *Node*, it will request the value from the other *Node* when it needs it.

Nodes that are in a logic graph but not connected to a valid *Flow* or have output *Ports* that are not connected to anything, will not contribute to the program and will not be exported, but they will be saved with the game data should you need to access them in the future.

There are several types of data that can be passed between logic nodes in Kwyll, each with a unique colour for the port and wires that connect them.

Number

Shown as green, a signed 16 bit integer value, capable of representing anything between -32768 and 32767.

Object

Shown as blue, a reference to an object in the game, can be a Room object or a global object.

Location

Shown as magenta, a reference to a location on the map.

Typically a *Port* will only allow connections from other *Ports* that have a matching type. However, some nodes do allow any type to be connected. Most notable among these are the various variable access nodes. This is because it is possible in Kwyll to store any value type in a variable, not just a numeric value. The facility allows you to store references to objects and locations in variables for future reference.

When two *Ports* of different types are connected, which Kwyll will only allow where it is valid, the wire connecting them will fade from one colour to the other to indicate that the value being passed is different to what is normally expected but is valid all the same.

Editing Nodes

See the Logic Editor for a detailed explanation of the tools for creating and editing logic graphs in Kwyll.

Nodes

Triggers

Always

The *Always Node* is a specific type of *Node* called a **Trigger**. A trigger *Node* is an entrypoint into a flow in the Kwyll logic. The *Always* trigger is the most commonly used trigger node, it will run the flow that follows on from it's **Flow Out** port every update of the game unless the *Skip* parameter is set to a non-zero positive value, in which case it will skip that many updates before running the flow, and then start again.

Ports

Flow Out

> As with all trigger nodes, an *Always Node* has only a single flow port on the output side. There is no input flow to a trigger node as it is the origin of a flow.

Parameters

Order

This is used to define the order in which multiple *Always Nodes* in a single logic program will be processed, in case there is some dependency between flows. If there are multiple *Always Nodes* with the same order, they will be processed together in an arbitrary order.

Skip

This parameter is used to delay the execution of the flow for a specified number of frames. If it is zero, the flow will run on every update as normal, any non-zero positive number will skip that many updates before running the flow.

Collided

The *Collided Node* is a specific type of *Node* called a **Trigger**. A trigger *Node* is an entrypoint into a flow in the Kwyll logic. The *Collided* trigger will get triggered when the object it is on has collided with the tilemap as a result of being moved with a Move Object node.

Ports

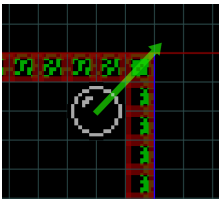
Flow Out

As with all trigger nodes, a *Collided Node* has only a single flow port on the output side. There is no input flow to a trigger node as it is the origin of a flow.

Sides

An integer output port that provides the sides that the collision occurred. Typically this will be a single side, but to accommodate the potential of simultaneously colliding with two sides, the value is encoded as a bit field. That is, the first 4 bits of the binary representation encode the up, down, left and right sides. Bit 0 is up, bit 1, down, bit 2 left and bit 3 right.

An Example:



In this image, the object is moving in the direction of the green arrow, which will result in it colliding in both right and up directions, as the tilemap has collision information indicating that the tiles cannot be passed in all directions, the resulting value will be bit 0 set and bit 3 set, bit 0 set results in a value of 1, bit 3 set results in a value of 8, add them together and the output of the Sides port would be 9.

+ Bits:

	0	**1**	**2**	**3**
SIDE	UP	DOWN	LEFT	RIGHT
VALUE	1	2	4	8

Initialise

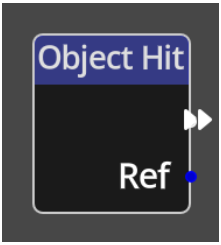
The *Initialise Node* is a specific type of *Node* called a **Trigger**. A trigger *Node* is an entrypoint into a flow in the Kwyll logic. The *Initialise* trigger will run the flow that follows on from it's **Flow Out** port only once at the start of the game.

Ports

Flow Out

> As with all trigger nodes, an *Initialse Node* has only a single flow port on the output side. There is no input flow to a trigger node as it is the origin of a flow.

Object Hit



The *Object Hit Node* is a specific type of *Node* called a **Trigger**. A trigger *Node* is an entrypoint into a flow in the Kwyll logic. The *Object Hit* trigger is triggered automatically when any object that is configured to detect intersections with other objects is moved into a position where it intersects another that is also flagged to detect intersections with other objects. Any defined *Object Hit* flows on both objects involved in the intersection will be triggered.

Ports

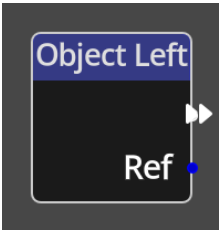
Flow Out

As with all trigger nodes, an *Animation Event Node* has only a single flow port on the output side. There is no input flow to a trigger node as it is the origin of a flow.

Ref

An object reference output port that provides a reference to the other object involved in the intersection.

Object Left



The *Object Left Node* is a specific type of *Node* called a **Trigger**. A trigger *Node* is an entrypoint into a flow in the Kwyll logic. The *Object Left* trigger is triggered automatically when two objects that were previously intersecting move into positions that mean they are no longer intersecting.

Ports

Flow Out

As with all trigger nodes, an *Animation Event Node* has only a single flow port on the output side. There is no input flow to a trigger node as it is the origin of a flow.

Ref

An object reference output port that provides a reference to the other object involved in the intersection.

On Message

The *On Message Node* is a specific type of *Node* called a **Trigger**. A trigger *Node* is an entrypoint into a flow in the Kwyll logic. The *On Message* trigger is triggered manually within another flow by using the Message Location, Message Object, Message Screen or Message Game Logic nodes.

A message consists of 3 parts, an ID and two optional parameters. The send message nodes have ports and parameters to set these values which control how this flow responds. The ID sent must match the ID defined in this flow for the flow to run. The parameters are typically used to control what the flow does. For example, a flow on a room that opens a door might take the side of the room that the door is on in P1 and the type of door in P2.

The text field at the bottom of this node can be used to give a name to this message flow, it is used for informational purposes only, as a reminder to the designer of what this flow is responsible for.

Ports

Flow Out

> As with all trigger nodes, an *On Message Node* has only a single flow port on the output side. There is no input flow to a trigger node as it is the origin of a flow. This flow is only followed if the ID of the message sent matches the ID of this flow.

P1

The value of the first parameter sent with the message, the *Message* nodes have ports and parameters to define this, so each use of the message flow can have different results based on the information passed in.

P2

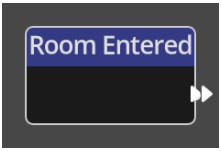
The value of the second parameter sent with the message, the *Message* nodes have ports and parameters to define this, so each use of the message flow can have different results based on the information passed in.

Parameters

ID

On Message Nodes have a single additional parameter, **ID**. This is the message ID that the flow will respond to. It is sent as part of the *Message* send nodes. Using this ID it is possible to have multiple flows on an object, room, screen or in the global logic that respond to different messages, multiple "functions" if you will.

Room Entered



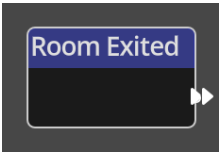
The *Room Entered Node* is a specific type of *Node* called a **Trigger**. A trigger *Node* is an entrypoint into a flow in the Kwyll logic. The *Room Entered* trigger will run the flow that follows on from it's **Flow Out** port upon entering a location that uses the room definition that this logic flow is defined on.

Ports

Flow Out

As with all trigger nodes, an *Room Entered Node* has only a single flow port on the output side. There is no input flow to a trigger node as it is the origin of a flow.

Room Exited



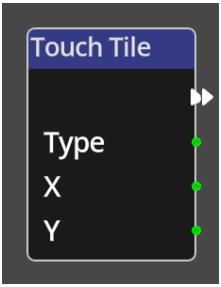
The *Room Exited Node* is a specific type of *Node* called a **Trigger**. A trigger *Node* is an entrypoint into a flow in the Kwyll logic. The *Room Exited* trigger will run the flow that follows on from it's **Flow Out** port upon exiting a location that uses the room definition that this logic flow is defined on.

Ports

Flow Out

As with all trigger nodes, an *Room Exited Node* has only a single flow port on the output side. There is no input flow to a trigger node as it is the origin of a flow.

Touch Tile



The *Touch Tile Node* is a specific type of *Node* called a **Trigger**. A trigger *Node* is an entrypoint into a flow in the Kwyll logic. The *Touch Tile* trigger will get triggered when a tile touched by the bounding rectangle of the object it is has a non-zero **Type** value.

Ports

Flow Out

As with all trigger nodes, a *Collided Node* has only a single flow port on the output side. There is no input flow to a trigger node as it is the origin of a flow.

Type

An integer output port that provides the type id of the tile that the ojbect is touching.

X

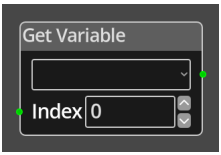
An integer output port that provides the X tile coordinate of the tile that the object is touching.

Y

An integer output port that provides the Y tile coordinate of the tile that the object is touching.

Variables

Get Variable



The *Get Variable Node* is used to get the integer value of a variable on whatever the current logic graph is running, i.e. a Screen, a Room, an Object, or the Game Logic.

It's important to understand that, in the case of rooms and objects the value being accessed is the value for the chosen variable in the Object Instance or Location, that is running the logic code, each instance has its own copy of the variables, they are independent.

Ports

Output

An integer output port that will provide the value of the chosen variable.

Index

An integer value for choosing which element in an Array variable to query. If the variable being accessed is not an Array variable, this parameter is ignored.

Parameters

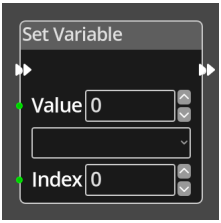
Variable

A dropdown listing the defined variables associated with the current logic graph.

Index

A constant integer for the Array index if the **Index** port is not connected.

Set Variable



The *Set Variable Node* is used to set the integer value of a variable on whatever the current logic graph is running, i.e. a Screen, a Room or an Object.

It's important to understand that, in the case of rooms and objects the value being modified is the value for the chosen variable in the Object Instance or Location, that is running the logic code, each reference has it's own version of the variables, so changing a variable on one reference for a particular object type will not affect the value on other references, they are independent.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Value

An integer input port used to provide the value that will be applied to

the chosen variable.

Parameters

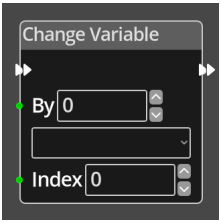
Value

A constant integer value to assign to the chosen variable. If the port is connected to another node, this option will become unavailable, the node will use the value provided on the input port.

Variable

A dropdown listing the defined variable names on the current logic graph.

Change Variable



The *Change Variable Node* is a simple way to modify the value of a variable on the current logic graph in one operation, avoiding the need for a Get, Math and Set sequence.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

By

An integer input port used to provide the amount that the variable will be changed by, this can be positive or negative.

Index

An integer value for choosing which element in an Array variable to change. If the variable being accessed is not an Array variable, this parameter is ignored.

Parameters

By

A constant integer amount that the variable will be changed by, used when the **By** port is not connected.

Variable

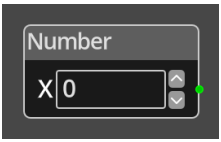
A dropdown listing the defined variables associated with the current logic graph.

Index

A constant integer for the Array index if the **Index** port is not connected.

Math

Number



The *Number Node* simply provides a constant fixed number to be used as the input to other nodes.

While most nodes offer constant parameters for integer input ports, this node can be useful when using the same input value for multiple nodes, as it allows you to change the value in one place rather than remembering to change it in multiple places, but still have the size and performance benefits of a constant value instead of a variable.

Ports

Out

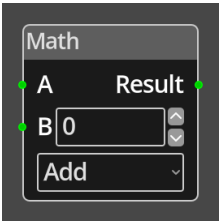
An integer output port that provides the specified constant value.

Parameters

Value

A constant integer value to provide.

Math



The *Math Node* takes input values A and B and applies a mathematical operation to them returning the result of the operation.

Ports

A

An integer input port used to provide the left operand of the mathematical operation.

B

An integer input port used to provide the right operand of the mathematical operation.

Result

An integer output port that provides the result of the mathematical operation.

Parameters

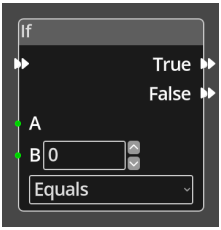
B

A constant integer value for the right operand, used when the **B** port is not connected.

Operator

The operation to perform on the two operands. A dropdown offers the choices, Add Subtract, Multiply and Divide.

If



The *If Node* performs a comparison between two input values and follows one of two flows depending on the result of the comparison.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

True

A node connected to the **True** port will be executed in sequence if the result of the comparison succeeds.

False

A node connected to the **False** port will be executed in sequence if the result of the comparison fails.

A

An integer input port used to provide the first value for comparison. This must be connected for the node to operate correctly.

B

An integer input port used to provide the second value for comparison.

Parameters

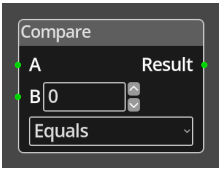
B

A constant integer value for the B operand, used when the **B** port is not connected.

Comparison

The type of comparison to perform, this can be any one of "equals", "greater than", "less than", "greater or equal", "less or equal", or "not equal".

Compare



The *Compare Node* performs a comparison between two input values and returns 0 if the comparison fails, or 1 if it succeeds.

Ports

A

An integer input port used to provide the first value for comparison. This must be connected for the node to operate correctly.

B

An integer input port used to provide the second value for comparison.

Result

The output of the comparison, either 0 if the comparison fails or 1 if it succeeds.

Parameters

B

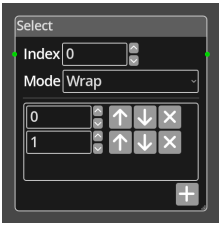
A constant integer value for the B operand, used when the **B** port is not connected.

Comparison

The type of comparison to perform, this can be any one of **Equals**,

Greater Than, **Less Than**, **Greater or Equal**, **Less or Equal**, or **Not Equal**.

Select



The *Select Node* takes an index value and returns the appropriate entry from a fixed list of integer values added to the node. Values are added using the plus button, and can be reordered and removed using the buttons next to each list entry.

The **Mode** parameter defines what the node will do if the provided index is out of range for the number of items in the list. See the Wrap and Clamp nodes for details of how the two modes work.

Ports

Index

An integer input port used to provide the index into the list.

Out

An integer output port that provides the value of the chose list item.

Parameters

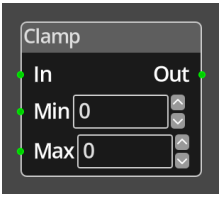
Index

A constant integer value for the index, used when the **Index** port is not connected.

Mode

The method to use to deal with out of range indices, **Wrap** or **Clamp**.

Clamp



The *Clamp Node* takes an integer value on the **In** port, if the value is less than or equal to the **Min** port/parameter the **Min** port/parameter value is output, if it is greater than or equal to the **Max** port/parameter the **Max** port/parameter value is output, otherwise the value is output unmodified.

Ports

In

An integer input port used to provide the input value for testing. This must be connected for the node to operate correctly.

Out

An integer output port, the result of the comparison detailed in the description is output on this port.

Min

An integer input port, using this allows you to specify the minimum value from elsewhere in the logic graph, such as via an Number node.

Max

An integer input port, using this allows you to specify the maximum value from elsewhere in the logic graph, such as via an Number node.

Parameters

Min

A constant integer value for the minimum, used when the **Min** port is not connected.

Max

A constant integer value for the maximum, used when the **Max** port is not connected.

Wrap

The *Wrap Node* takes an integer value on the **In** port, if the value is less than 0, it will negate the value and subtract that from the **Max** value minus 1, and repeat until it is greater than or equal to 0.

If the input value is greater than or equal to the **Max** value, it will subtract the **Max** value, and repeat until it is less than the **Max** value.

Ports

In

An integer input port used to provide the input value for testing. This must be connected for the node to operate correctly.

Out

An integer output port, the result of the comparison detailed in the description is output on this port.

Max

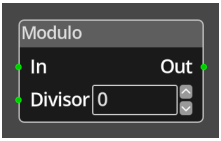
An integer input port, using this allows you to specify the maximum value from elsewhere in the logic graph, such as via an Number node.

Parameters

Max

A constant integer value for the maximum, used when the **Max** port is not connected.

Modulo



The *Modulo Node* takes an integer value on the **In** port, and another integer value on the **Divisor** port and returns the remainder of dividing the **In** value by the **Divisor** value.

Ports

In

An integer input port used to provide the input value for the calculation.

Out

An integer output port, the remainder of the division detailed in the description is output on this port.

Divisor

An integer value to divide the input by to get the remainder.

Parameters

In

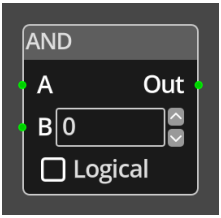
A constant integer value for the input, used when the **In** port is not connected.

Divisor

A constant integer value for the divisor, used when the **Divisor** port is

not connected.

And



The *And Node* takes two inputs of type integer and performs either a logical or bitwise AND operation on them before providing the result on the *Out* port.

If the *Logical* flag is set, the node will consider any non-zero value as "true" and 0 as false for both parameters, and output a 1 if both are true otherwise 0.

If the *Logical* flag is clear, the node will peform a bitwise AND on the two 16 bit integer parameters, resulting in a 16 bit integer where each bit is set only if both parameters have that same bit set.

Ports

A and B

The two input values, they must be linked for the node to be operational and can only be connected to sources that provide an integer value, such as Get.

Out

The output of the node.

Parameters

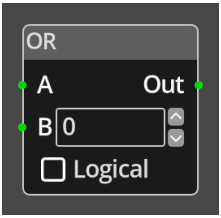
B

A constant integer value for the right operand, used when the **B** port is not connected.

Logical

This defines whether the AND is to be performed as a logical or bitwise AND.

Or



The *Or Node* takes two inputs of type integer and performs either a logical or bitwise OR operation on them before providing the result on the *Out* port.

If the *Logical* flag is set, the node will consider any non-zero value as "true" and 0 as false for both parameters, and output a 1 if either are true otherwise 0.

If the *Logical* flag is clear, the node will peform a bitwise OR on the two 16 bit integer parameters, resulting in a 16 bit integer where each bit is set if either parameters have that same bit set.

Ports

A and B

The two input values, they must be linked for the node to be operational and can only be connected to sources that provide an integer value, such as Get.

Out

The output of the node.

Parameters

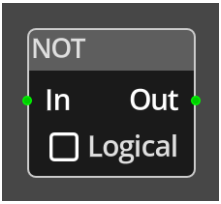
B

A constant integer value for the right operand, used when the **B** port is not connected.

Logical

This defines whether the OR is to be performed as a logical or bitwise OR.

Not



The *Not Node* takes a single input of type integer and performs either a logical or bitwise NOT operation on it before providing the result on the *Out* port.

If the *Logical* flag is set, the node will consider any non-zero value as "true" and 0 as false, and output a 1 if the input is false otherwise 0.

If the *Logical* flag is clear, the node will peform a bitwise NOT on the 16 bit integer input, resulting in a 16 bit integer where each bit is set if the bit is unset in the input.

Ports

n

The input value, must be linked for the node to be operational and can only be connected to sources that provide an integer value, such as Get.

Out

The output of the node.

Parameters

Logical

This defines whether the NOT is to be performed as a logical or bitwise NOT.

Random

The *Random Node* provides a single random number between the **Min** and **Max** values provided.

Ports

Min

The minimum value, inclusive, of the random number.

Max

The maximum value, inclusive, of the random number.

Value

An integer output port that provides the generated random number.

Parameters

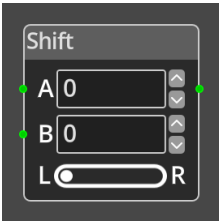
Min

A constant integer value for the minimum, used when the **Min** port is not connected.

Y

A constant integer value for the maximum, used when the **Max** port is not connected.

Shift



The *Shift* node allows you to shift the bits of a value left or right by any number of bits. Two common uses of the shift operator are to implement bitwise operations, usually when combined with the And, Or and Not, nodes. to access and use single bits of an integer value or variable, and to perform cheap multiply and divide operations when multiplying or dividing by a value that is a power of 2, i.e. 2, 4, 8, 16 etc. Shifting left in binary effectively multiplies by 2, while shifting right divides by 2, but is much cheaper than a multiply or divide.

Ports

A

The value to shift, a numerical integer value.

B

The amount to shift the value by, in bits.

Parameters

A

A constant integer value for value, used when the **A** port is not connected.

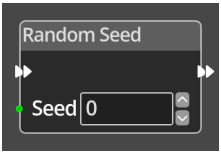
A

A constant integer value for shift amount, used when the **B** port is not connected.

L/R

The direction to shift, Left, or Right.

Random Seed



The *Random Seed Node* is used to provide a starting point for the Random node to generate random numbers from. When providing a seed value for the random number generator, you can be certain that the sequence of random numbers from that point on will be repeatable. This allows you to use random values, but with predictability.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Seed

The integer value to seed the random number generator with.

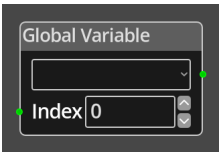
Parameters

Seed

A constant integer value for the seed, used when the **Seed** port is not connected.

Global

Global Variable



The *Global Variable Node* is used to get the integer value of a variable on the global Logic.

Ports

Output

An integer output port that will provide the value of the chosen variable.

Index

An integer value for choosing which element in an Array variable to change. If the variable being accessed is not an Array variable, this parameter is ignored.

Parameters

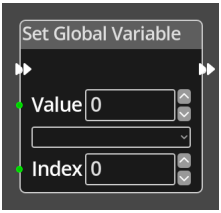
Variable

A dropdown listing the defined variables associated with the global logic graph.

Index

A constant integer for the Array index if the **Index** port is not connected.

Set Global Variable



The *Set Global Variable Node* is used to set the integer value of a variable on the Global Logic.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Value

An integer input port used to provide the value that will be applied to the chosen variable.

Index

An integer value for choosing which element in an Array variable to change. If the variable being accessed is not an Array variable, this parameter is ignored.

Parameters

Value

A constant integer value to assign to the chosen variable. If the port is connected to another node, this option will become unavailable, the node will use the value provided on the input port.

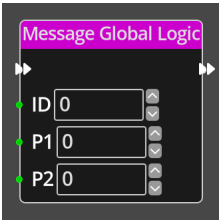
Variable

A dropdown listing the defined variable names on the global logic graph.

Index

A constant integer for the Array index if the **Index** port is not connected.

Message Global Logic



The *Message Global Logic Node* sends a message to the global logic in your game. The global logic should have an On Message node correctly configured to handle the message, otherwise it will have no effect.

The ID of the message is used to choose which On Message flow will handle the message, and the optional parameters P1 and P2 are passed to the message handler to provide additional information that can be arbitrarily defined by the game designer.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

ID

An integer input port used to provide ID of the message, this controls which flow on the global object will handle the message, only On Message flows with a matchin ID will be executed.

P1

An integer input port used to provide an arbitrary parameter to the handler, the purpose of which is defined by the game designer.

P2

An integer input port used to provide an arbitrary parameter to the handler, the purpose of which is defined by the game designer.

Parameters

ID

A constant integer value for the message ID, used when the **ID** port is not connected.

P1

A constant integer value for the optional P1 parameter, used when the **P1** port is not connected.

P2

A constant integer value for the optional P2 parameter, used when the **P2** port is not connected.

Get Global Object

The *Get Global Object Node* is used to get an Object Instance to a global object for use further down the graph in other nodes that accept a *Ref* input. This allows you to directly affect global objects, moving them, etc.

Ports

Ref

An Object Ref output port that will provide the reference to the requested global object.

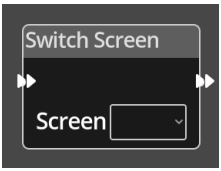
Parameters

Object

A dropdown selector that will list all the currently available global objects. The value selected will dictate which Ref is returned.

Screen

Switch Screen



The *Switch Screen Node* changes the current screen to the specified one.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

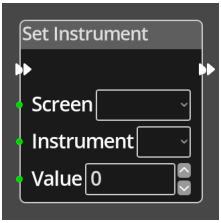
A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Parameters

Screen

The screen to switch to.

Set Instrument



The *Set Instrument Node* sets the value of an instrument on a screen. The screen is chosen from a fixed list of screens, the instrument is referenced by an index into the list of instruments on that screen.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Instrument

An integer input port used to provide the index of the instrument on the screen.

Value

An integer input port used to provide the value to apply to the instrument.

Parameters

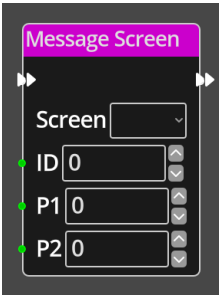
Instrument

A constant integer value for the instrument index, used when the **Instrument** port is not connected.

Value

A constant integer value for the value, used when the **Value** port is not connected.

Message Screen



The *Message Screen Node* sends a message to specified screen. The screen logic should have an On Message node correctly configured to handle the message, otherwise it will have no effect.

The ID of the message is used to choose which On Message flow will handle the message, and the optional parameters P1 and P2 are passed to the message handler to provide additional information that can be arbitrarily defined by the game designer.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

ID

An integer input port used to provide ID of the message, this controls

which flow on the global object will handle the message, only On Message flows with a matching ID will be executed.

P1

An integer input port used to provide an arbitrary parameter to the handler, the purpose of which is defined by the game designer.

P2

An integer input port used to provide an arbitrary parameter to the handler, the purpose of which is defined by the game designer.

Parameters

Screen

The screen to send the message to. A dropdown will show a list of available screens to select from.

ID

A constant integer value for the message ID, used when the **ID** port is not connected.

P1

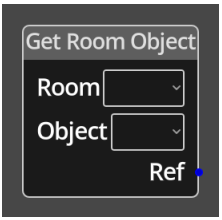
A constant integer value for the optional P1 parameter, used when the **P1** port is not connected.

P2

A constant integer value for the optional P2 parameter, used when the **P2** port is not connected.

Room

Get Room Object



The *Get Room Object Node* is used to get an Object Instance to an object in a room for use further down the graph in other nodes that accept a *Ref* input. This allows you to directly affect room objects, moving them, etc.

Ports

Ref

An Object Ref output port that will provide the reference to the requested global object.

Parameters

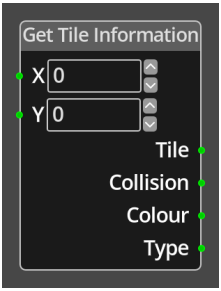
Room

A dropdown selector that will list all the currently available rooms. When a room is selected, the contents of the **Object** selector will update to list the object references in that room.

Object

A dropdown selector that will list all the currently available objects in the selected room. The value selected will dictate which Ref is returned.

Get Tile Information



The *Get Tile Information Node* takes input values X and Y in tile coordinates and provides back information about the tile at that location in the room tilemap.

NOTE: This tile only works for the room tilemap, not the screen tilemap.

Ports

X

An integer input port used to provide the X value of the tile coordinate.

Y

An integer input port used to provide the Y value of the tile coordinate.

Tile

An integer output port, providing the tile number at the specified coordinates if there is a tile there. If there is no tile, the output value with be -1.

Collision

An integer output port providing the collision flags at the specified coordinates. The value is a bit field representing the sides of the tile that are marked as preventing object movement. See collision.

Colour

An integer output port providing the colour information at the specified coordinates. The colour is a combination of ink, paper and bright data as described in colour.

Type

An integer output port providing the custom type of the tile at the specified coordinates. If there is no tile at the specified location, the output value will be 0. See tilemaps for details of tile types.

Parameters

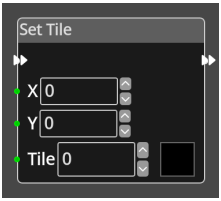
X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Set Tile



The *Set Tile Node* is used to set the current tile type drawn into a given cell on the Tilemap in the current room.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

X

An integer input port used to provide the X value of the tile coordinate.

Y

An integer input port used to provide the Y value of the tile coordinate.

Tile

An integer input port used to provide the tile index to change the specified tilemap cell to.

Parameters

X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Tile

A constant integer value for the tile index, used when the *Tile*port is not connected.

Set Tile Colour

The *Set Tile Colour Node* is used to set the current tile type drawn into a given cell on the Tilemap in the current room.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

X

An integer input port used to provide the X value of the tile coordinate.

Y

An integer input port used to provide the Y value of the tile coordinate.

Ink

An integer input port used to provide ink colour to apply to the given

cell, see Colour. This can be set to "X" to leave the current tile ink unaltered.

Paper

An integer input port used to provide paper colour to apply to the given cell, see Colour. This can be set to "X" to leave the current tile paper unaltered.

Bright

An integer input port used to provide paper colour to apply to the given cell, see Colour, a 0 value is not bright, any non-zero value is bright. This can be set to "Leave" to leave the current tile brightness unaltered.

Parameters

X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Ink

A constant integer value for the the ink colour, used when the **Ink** port is not connected.

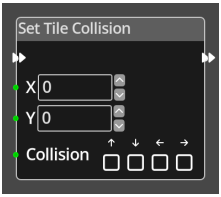
Paper

A constant integer value for the the paper colour, used when the **Paper** port is not connected.

Bright

A constant integer value for the brightness setting to assign at the specified coordinates, used when the **Bright** port is not connected.

Set Tile Collision



The *Set Tile Collision Node* is used to set which sides of the given cell on the Tilemap in the current room, are impenetrable. See Tilemap Collision.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

X

An integer input port used to provide the X value of the tile coordinate.

Y

An integer input port used to provide the Y value of the tile coordinate.

Collision

An integer input port used to provide collision sides to apply to the given cell.

Parameters

X

A constant integer value for the X coordinate, used when the **X** port is not connected.

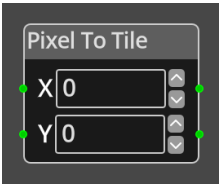
Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Collision

A constant value for the collision sides, used when the **Collision** port is not connected.

Pixel To Tile



The *Pixel To Tile Node* takes input values X and Y that specify a position in pixel coordinates and converts them into tile coordinates.

The coordinate system of the input, room or map, is not important, the conversion simply takes the pixel coordinate and returns the tile coordinate for the pixel at the position, irrespective of whether it is on screen, in the room, or not.

See Coordinate Systems for more information about how coordinate systems work in Kwyll and how to convert between the various systems in logic.

Ports

X

An integer input port used to provide the X value of the pixel coordinate.

Y

An integer input port used to provide the Y value of the pixel coordinate.

X Out

An integer output port that provides the converted X value in tile coordinates.

Y Out

An integer output port that provides the converted Y value in tile coordinates.

Parameters

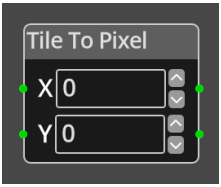
X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Tile To Pixel



The *Tile To Pixel Node* takes input values X and Y that specify a position in tile coordinates and converts them into pixel coordinates.

The coordinate system of the input, room or map, is not important, the conversion simply takes the tile coordinate and returns the pixel coordinate for the top left pixel of the tile, irrespective of whether it is on screen, in the room, or not.

See Coordinate Systems for more information about how coordinate systems work in Kwyll and how to convert between the various systems in logic.

Ports

X

An integer input port used to provide the X value of the tile coordinate.

Y

An integer input port used to provide the Y value of the tile coordinate.

X Out

An integer output port that provides the converted X value in pixel coordinates.

Y Out

An integer output port that provides the converted Y value in pixel

coordinates.

Parameters

X

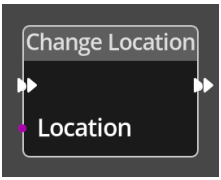
A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Map

Change Location



The *Change Location Node* is used to switch to another Location on the map.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

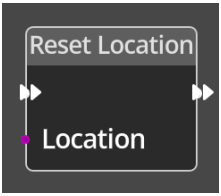
Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Location

A location reference input port that provides a reference to the location to change to.

Reset Location



The *Reset Location* node resets the specified location, the current location if not provided. This operation returns all room objects to their initial position, runs the Initialise flows on all room objects, and resets all variables and then runs the Initialise flow on the location and resets its variables to their initial values.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

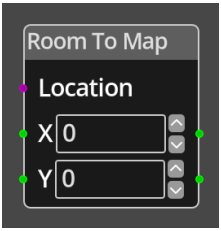
Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Location

A location reference input port that provides a reference to the location to reset.

Room To Map



The *Room To Map Node* takes input values X and Y that define a position in the room coordinates of a specified Location and converts the position into map coordinates.

This is particularly useful for positioning a map level object so that it is at a chosen point within a location.

See Coordinate Systems for more information about how coordinate systems work in Kwyll and how to convert between the various systems in logic.

Ports

Location

A location reference port that provides a reference to the location that the provided coordinates are relative to. If not connected, and the logic is on a location, the current location will be used.

X

An integer input port used to provide the X value of the location coordinate.

Y

An integer input port used to provide the Y value of the location coordinate.

X Out

An integer output port that provides the X value of the position in map coordinates.

Y Out

An integer output port that provides the Y value of the position in map coordinates.

Parameters

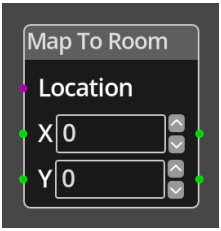
X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Map To Room



The *Map To Room Node* takes input values X and Y that define a position in map coordinates and converts the position to the room coordinates of a specified Location.

This is particularly useful for positioning a room level object so that it is at a chosen position on the map, presuming the position is within the bounds of the location.

See Coordinate Systems for more information about how coordinate systems work in Kwyll and how to convert between the various systems in logic.

Ports

Location

A location reference port that provides a reference to the location that the resulting coordinates are relative to. If not connected, and the logic is on a location, the current location will be used.

X

An integer input port used to provide the X value of the map coordinate.

Y

An integer input port used to provide the Y value of the map

coordinate.

X Out

An integer output port that provides the X value of the position in room coordinates.

Y Out

An integer output port that provides the Y value of the position in room coordinates.

Parameters

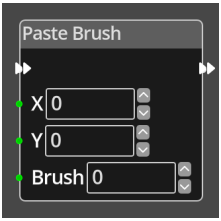
X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Paste Brush



The *Paste Brush Node* takes input values X and Y that specify a position on the room tilemap in tile coordinates and will paste the contents of the brush at the specified index into that location on the tilemap.

Pasting brushes only applies to Room tilemaps.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

X

An integer input port used to provide the X value of the tilemap coordinate.

Y

An integer input port used to provide the Y value of the tilemap coordinate.

Brush

An integer input port used to specify the index of the brush to paste at the given position.

Parameters

X

A constant integer value for the X coordinate, used when the **X** port is not connected.

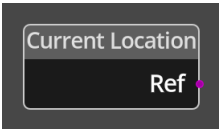
Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Brush

A constant integer value for the brush index, used when the **Brush** port is not connected.

Current Location



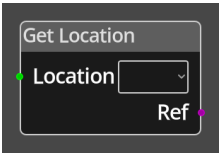
The *Current Location Node* simply returns a reference to the current location on the map for use in other nodes or for storage in a variable for future reference.

Ports

Out

A location reference output port, provides a reference to the current location on the map.

Get Location



The *Get Location Node* takes an integer value on the **Location** port and returns a reference to the location with that index on the map. The map editor displays the location index in the top left corner of each location.

Ports

Location

An integer input port used to provide the index of the location to get a reference to.

Ref

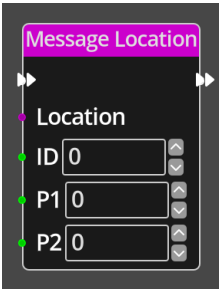
A location reference that points to the given location on the map.

Parameters

Location

A constant value for the location index. The user will be able to choose from a drop down list of the existing location names, which will be stored internally as an index, for convenience.

Message Location



The *Message Location Node* sends a message to a specified location. The room logic for the location should have an On Message node correctly configured to handle the message, otherwise it will have no effect.

The ID of the message is used to choose which On Message flow will handle the message, and the optional parameters P1 and P2 are passed to the message handler to provide additional information that can be arbitrarily defined by the game designer.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Location

A location reference port that provides a reference to the location that

the message will be sent to. If not connected, and the logic is on a location, the current location will be used.

ID

An integer input port used to provide ID of the message, this controls which flow on the global object will handle the message, only On Message flows with a matchin ID will be executed.

P1

An integer input port used to provide an arbitrary parameter to the handler, the purpose of which is defined by the game designer.

P2

An integer input port used to provide an arbitrary parameter to the handler, the purpose of which is defined by the game designer.

Parameters

ID

A constant integer value for the message ID, used when the **ID** port is not connected.

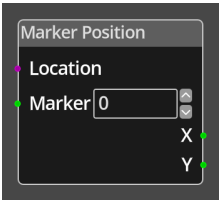
P1

A constant integer value for the optional P1 parameter, used when the **P1** port is not connected.

P2

A constant integer value for the optional P2 parameter, used when the **P2** port is not connected.

Marker Position



The *Marker Position Node* queries the position of a Marker in a given location, returning the position in room coordinates.

Ports

Location

A location reference port that provides a reference to the location that the provided coordinates are relative to. If not connected, and the logic is on a location, the current location will be used.

Marker

An integer input port that provides the index of the marker to query.

X

An integer output port that provides the X value of the marker coordinate.

Y

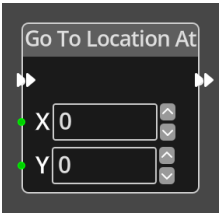
An integer output port that provides the Y value of the marker coordinate.

Parameters

Marker

A constant integer value for the marker index, used when the **Marker** port is not connected.

Go To Location At



The *Go To Location At Node* takes input values X and Y that specify a position on the map and will change to the Location at that point on the map if there is one. If there is no location at the specified coordinates, nothing happens.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

X

An integer input port used to provide the X value of the map coordinate.

Y

An integer input port used to provide the Y value of the map coordinate.

Parameters

X

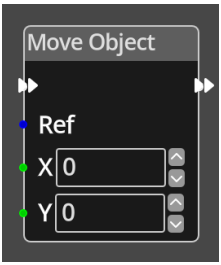
A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Object

Move Object



The *Move Object Node* takes input values X and Y that specify a position to move the specified object to.

When moving an object using this node, if the object has the *Collide Bg* flag enabled, the node will check if the movement results in a collision with the background tilemap and prevent it, resulting in a different endpoint to the requested one.

The X and Y coordinates are in the coordinate system that is appropriate for the object being moved. If the object is a room object, it expects the position to be in room coordinates, if it is a map object, it expects it to be in map coordinates.

Ports

Flow In

> In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Ref

An object reference port that provides a reference to the object to check. If this is not connected, and the logic is on an object, the current object will be used.

X

An integer input port used to provide the X value of the new position.

Y

An integer input port used to provide the Y value of the new positio.

Parameters

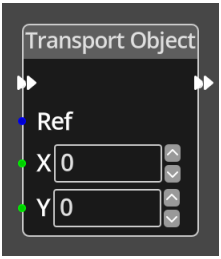
X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Transport Object



The *Transport Object Node* takes input values X and Y that specify a position to transport the specified object to.

Transporting an object differs to the function of the Move Object node in that it performs no intersection testing or collision detection, it simply moves it directly to the chosen position, as such, it is very fast and should be preferred for moving objects when intersection and collision are not important.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Ref

An object reference port that provides a reference to the object to transport. If this is not connected, and the logic is on an object, the

current object will be used.

X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Parameters

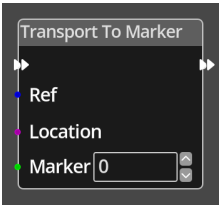
X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

Transport To Marker



The *Transport To Marker Node* queries the position of a Marker in a given location, and immediately transports the given object to that location, taking into account whether the object is a room object or a map object.

Ports

Ref

An object reference port that provides a reference to the object to transport. If this is not connected, and the logic is on an object, the current object will be used.

Location

A location reference port that provides a reference to the Location contains the marker to query. If not connected, and the logic is on a location, the current location will be used.

Marker

An integer input port that provides the index of the marker to query.

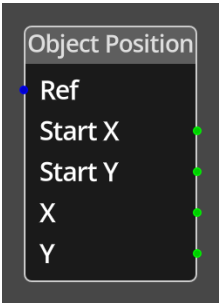
Parameters

Marker

A constant integer value for the marker index, used when the **Marker**

port is not connected.

Object Position



The *Object Position Node* provides position information about the current Object Instance that the logic graph is running on.

The position data is returned in the appropriate coordinate system for the object being referenced. A global or dynamic object will return coordinates in "map" space, a room object will return coordinates in the local space of the room.

Ports

Ref

An Object Reference input port that can provide an object that will be used to lookup the instance data. If this is not connected and the logic is on an Object, the current object will be used instead. If the logic is not on an object, 0 will be returned on all output ports.

Start X

The initial X position of the object, this is constant, and used to reset the object's position at the start of the game.

Start Y

The initial Y position of the object, this is constant, and used to reset the object's position at the start of the game.

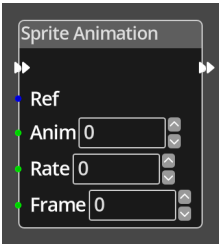
X

The current X position of the object, this is the live position of the object as set by any Move Object nodes, or other nodes that can modify an object's position.

Y

The current Y position of the object, this is the live position of the object as set by any Move Object nodes, or other nodes that can modify an object's position.

Sprite Animation



The *Sprite Animation Node* is used to modify the settings for any sprite animations created on an object instance. It can optionally change the current animation, the rate and the frame in the current or new animation.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Ref

An object reference port that provides a reference to the object to check. If this is not connected, and the logic is on an object, the current object will be used.

Anim

An integer port that provides the index of the animation to switch to. If the value is -1, the current animation is not changed.

Rate

An integer port that provides the requested animation rate for the sprite animation. If the value is -1, the rate is not modified. A value of 0 effectively stops the sprite animation from playing. The rate is a delay between changing animation frame in game frames. If the game is successfully running at 50fps, a value of 50 will make the animation play at one frame per second.

Frame

An integer port that provides the requested frame number on the current animation. If the value is -1, the frame is not modified, unless the animation has changed, in which case, to avoid potential out of range frame numbers, the frame is reset to

1. If you want to switch to another animation on a specific frame, you must give a positive frame number here.

Parameters

Anim

A constant integer value for the animation index, used when the **Anim** port is not connected.

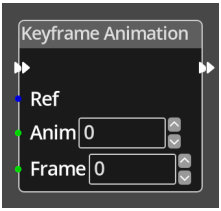
Rate

A constant integer value for the animation rate, used when the **Rate** port is not connected.

Frame

A constant integer value for the animation frame, used when the **Frame** port is not connected.

Keyframe Animation



The *Keyframe Animation Node* is used to modify the settings for any keyframe animations created on an object instance, currently only available for Room Objects. It can optionally change the current animation and/or the frame in the current or new animation.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Ref

An object reference port that provides a reference to the object to check. If this is not connected, and the logic is on an object, the current object will be used.

Anim

An integer port that provides the index of the animation to switch to. If the value is -1, the current animation is not changed.

Frame

An integer port that provides the requested frame number on the current animation. If the value is -1, the frame is not modified, unless a change in animation makes the current frame number invalid.

Parameters

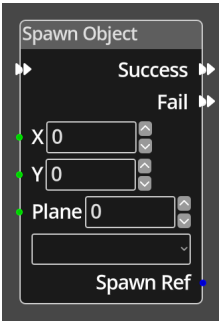
Anim

A constant integer value for the animation index, used when the **Anim** port is not connected.

Frame

A constant integer value for the animation frame, used when the **Frame** port is not connected.

Spawn Object



The *Spawn Object Node* attempts to spawn a new dynamic object into the game. It takes an object definition, and if there is enough dynamic object capacity left, will create a new instance of that object definition as a map level dynamic object at the position in map coordinates specified in the **X** and **Y** values.

If the operation succeeds, the flow will follow the **Success** flow, if it fails due to there being insufficient dynamic object slots available, it will follow the **Fail** flow.

If the spawn operation succeeds, the **Spawn Ref** port will provide a reference to the spawned object for further modification or recording in a variable. If the spawn fails, this port value is undefined and should not be used.

Ports

Flow In

> In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of

any other flow node.

Success

A node connected to the **Success** port will be executed in sequence following the successful completion of this node's operation.

Fail

A node connected to the **Fail** port will be executed in sequence if the operation failed.

X

An integer input port used to provide the X value of the map coordinate.

Y

An integer input port used to provide the Y value of the map coordinate.

Plane

An integer input port used to provide the plane on which the object's sprite will render. See Planes for more information about sprite planes.

Spawn Ref

An object reference output port that provides a reference to the spawned object if the operation succeeded.

Parameters

X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is

not connected.

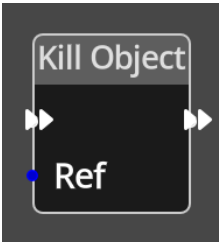
Plane

A constant integer value for the plane, used when the **Plane** port is not connected

Object Definition

A dropdown will provide a list of possible object types to choose from to spawn.

Kill Object



The *Kill Object Node* is used to remove a dynamic object that was created with the Spawn Object node.

It is not possible to kill map or room objects, only those created with the Spawn Object node.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

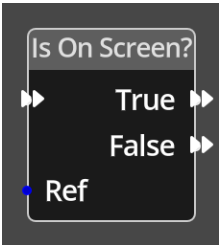
Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Ref

An object reference port that provides a reference to the object to check. If this is not connected, and the logic is on an object, the current object will be used.

Is On Screen?



The *Is On Screen Node* takes a reference to an object and checks if its position puts it in a potentially visible place in the game window. It will continue executing in one of the **True** or **False** paths depending on the outcome of the test.

Potentially visible means in a location that is within the bounds of the game window, the object could be on screen, but not visible if the visible flag is not set.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Ref

An object reference port that provides a reference to the object to check. If this is not connected, and the logic is on an object, the current object will be used.

True

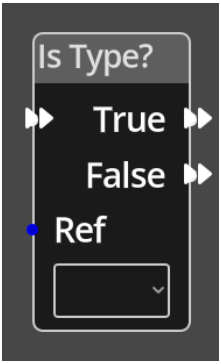
A node connected to the **True** port will be executed in sequence if the

object is in a potentially visible position in the window.

False

A node connected to the **False** port will be executed in sequence if the object is not in a potentially visible position in the window.

Is Type?



The *Is Type Node* takes a reference to an object and checks if it uses the given object definition. It will continue executing in one of the **True** or **False** paths depending on the outcome of the test.

Ports

Flow In

> In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Ref

> An object reference port that provides a reference to the object to check. If this is not connected, and the logic is on an object, the current object will be used.

True

> A node connected to the **True** port will be executed in sequence if the object uses the specified object definition.

False

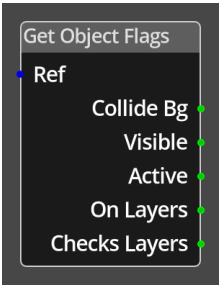
A node connected to the **False** port will be executed in sequence if the object does not use the specified object definition.

Parameters

Object Definition

A dropdown to select the Object Definition that will be compared with the Object Definition the specified object is based on.

Get Object Flags



The *Get Object Flags Node* allows the user to query the various flags on an object in the game. As the flags on an object are represented as bits in a larger value, the actual number returned is the value of that bit in the flags, as detailed below for each output port.

Ports

Ref

An object reference port that provides a reference to the object to query the flags on. If this is not connected, and the logic is on an object, the current object will be used.

Collide Bg

The value of the collides with background flag. This is bit 1, if the flag is set the result will be 2.

Visible

The value of the visible flag. This is bit 3, if the flag is set the result will be 8.

Active

The value of the active flag. This is bit 4, if the flag is set the result will be 16.

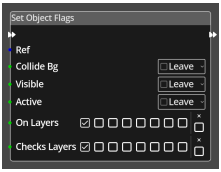
On Layers

A bit field representing the intersection layers this object belongs to, bits 0 to 7 represent layers 0 to 7.

Checks Layers

A bit field representing the intersection layers this object will check for intersections with, bits 0 to 7 represent layers 0 to 7.

Set Object Flags



The *Set Object Flags Node* is used to modify the flags on a given object. It can modify the flags that control whether the object registers intersection with other objects during moving, whether the object collides with the bacground tilemap or not, whether it is visible, and whether it is active.

The *Set Object Flags Node* is slightly different to many nodes in regards to how the parameters are set. This is due to the fact that it can be used to only affect certain flags, and leave others unchanged. To accommodate this, each flag is represented by a "three state checkbox". The three states are "on", "off", and "leave". If any are set to **Leave**, that flag will not be modified from its current state.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Ref

An object reference port that provides a reference to the object to change ths flags of. If not connected, and the logic is on an object, the

current object will be used.

Collide Bg

An integer input port used to provide the value of the *Collide Bg* flag, 0 is off, any other value is on.

Visible

An integer input port used to provide the value of the *Visible* flag, 0 is off, any other value is on.

Active

An integer input port used to provide the value of the *Active* flag, 0 is off, any other value is on.

On Layers

A bit field representing the intersection layers this object belongs to, bits 0 to 7 represent layers 0 to 7.

Checks Layers

A bit field representing the intersection layers this object will check for intersections with, bits 0 to 7 represent layers 0 to 7.

Parameters

Intersect Objects

A constant value for the *Intersect Objects* flag, used when the **Intersect Objects** port is not connected. If the value is **Leave**, the flag state will not be changed.

Collide Bg

A constant value for the *Collide Bg* flag, used when the **Collide Bg** port is not connected. If the value is **Leave**, the flag state will not be changed.

Visible

A constant value for the *Visible* flag, used when the **Visible** port is not connected. If the value is **Leave**, the flag state will not be changed.

Active

A constant value for the *Active* flag, used when the **Active** port is not connected. If the value is **Leave**, the flag state will not be changed.

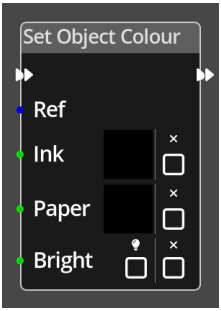
Ignore On Layers

(checkbox labelled 'x' next to the On Layers settings) If this flag is set, no changes will be made to the "On Layers" property of the object.

Ignore Checks Layers

(checkbox labelled 'x' next to the Checks Layers settings) If this flag is set, no changes will be made to the "Checks Layers" property of the object.

Set Object Colour



The *Set Object Colour Node* sets the colour information for the given object.

The *Set Object Colour Node* is a little different to many other nodes in the way it presents parameters, this is due to the fact that it can be used to modify only certain properties of the object colour. The primary difference in the parameters is the inclusion of an additional checkbox for each with a small **X** above it, this check box is used to indicate that the operation should ignore that property altogether. For example, selecting the **X** checkbox on the **Ink** property would ensure that the operation makes no changes at all to the ink colour of the object.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Ref

An object reference port that provides a reference to the object to change the colours of. If not connected, and the logic is on an object, the current object will be used.

Ink

An integer input port used to provide the ink colour index.

Paper

An integer input port used to provide the paper colour index.

Bright

An integer input port used to provide the brightness setting, 0 is non-bright, any other value is bright.

Parameters

Ink

A colour selector to choose visually the ink colour to use for the object. The checkbox with the **X** above indicates that the ink colour of the object should not be changed.

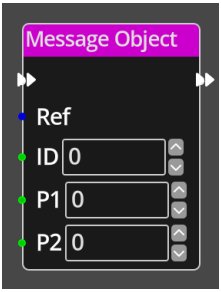
Paper

A colour selector to choose visually the paper colour to use for the object. The checkbox with the **X** above indicates that the paper colour of the object should not be changed.

Bright

A checkbox to select whether the object will be bright or not. The checkbox with the **X** above indicates that the brightness of the object should not be changed.

Message Object



The *Message Object Node* sends a message to a specified object. The object logic should have an On Message node correctly configured to handle the message, otherwise it will have no effect.

The ID of the message is used to choose which On Message flow will handle the message, and the optional parameters P1 and P2 are passed to the message handler to provide additional information that can be arbitrarily defined by the game designer.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Ref

An object reference port that provides a reference to the object to

check. If this is not connected, and the logic is on an object, the current object will be used.

ID

An integer input port used to provide ID of the message, this controls which flow on the global object will handle the message, only On Message flows with a matchin ID will be executed.

P1

An integer input port used to provide an arbitrary parameter to the handler, the purpose of which is defined by the game designer.

P2

An integer input port used to provide an arbitrary parameter to the handler, the purpose of which is defined by the game designer.

Parameters

ID

A constant integer value for the message ID, used when the **ID** port is not connected.

P1

A constant integer value for the optional P1 parameter, used when the **P1** port is not connected.

P2

A constant integer value for the optional P2 parameter, used when the **P2** port is not connected.

IO

Key Input

The *Key Input Node* reports on the status of a specific single key, or any key.

The node reports the status in one of four modes, *Pressed*, *Released*, *Just Pressed* and *Just Released*. The *Pressed* and *Released* modes will report continuously while the input is in the appropriate state, either pressed or not pressed. The *Just Pressed* and *Just Released* will report only for the frame after the input changes to the appropriate state, pressed or released. For example, *Just Pressed* will report only once when the player presses the key or joystick control, commonly used for fire actions, it will not report again until the player releases that input and then activates it again.

Ports

Output

An integer output port that will report the result of checking the state of the key based on the mode. If node is configured to check for a specific key, the output will be 0 or 1. If the node is configured to check for any key, the output will be the Keycode of the first key that is found to be in the configured state, *Pressed*, *Released*, *Just Pressed* or *Just Released*.

Parameters

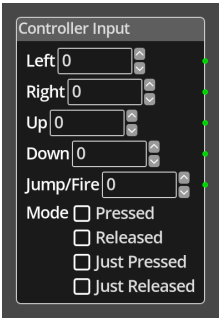
Key

A dropdown list of keys you can check for, or "ANY" to check for any key.

Mode

Define the mode of operation between *Pressed*, *Released*, *Just Pressed* and *Just Released*. See above for an explanation of the difference between the modes.

Controller Input



The *Controller Input Node* reports on the status of the configured controller for left, right, up, down and jump/fire. This node will adapt to whatever has been configured for the input method in your game, joystick, keyboard etc. The node allows you to define which values to return when each input is activated, defaulting to 1 for each, so that you can define the appropriate values for your needs. For example, it might be common to use -1 for left and 1 for right, to be used directly later in the graph as directions on the X axis to avoid having to convert the input to a direction manually.

The controller reports the status in one of four modes, *Pressed*, *Released*, *Just Pressed* and *Just Released*. The *Pressed* and *Released* modes will report continuously while the input is in the appropriate state, either pressed or not pressed. The *Just Pressed* and *Just Released* will report only for the frame after the input changes to the appropriate state, pressed or released. For example, *Just Pressed* will report only once when the player presses the key or joystick control, commonly used for fire actions, it will not report again until the player releases that input and then activates it again.

Ports

Left

An integer output port that will report the result of checking the state of the left controller input based on the mode.

Right

An integer output port that will report the result of checking the state of the right controller input based on the mode.

Up

An integer output port that will report the result of checking the state of the up controller input based on the mode.

Down

An integer output port that will report the result of checking the state of the down controller input based on the mode.

Jump/Fire

An integer output port that will report the result of checking the state of the jump/fire controller input based on the mode.

Parameters

Left

A constant integer value that will be output on the **Left** port if the input check matches the mode for the left input.

Right

A constant integer value that will be output on the **Right** port if the input check matches the mode for the right input.

Up

A constant integer value that will be output on the **Up** port if the input check matches the mode for the up input.

Down

A constant integer value that will be output on the **Down** port if the input check matches the mode for the down input.

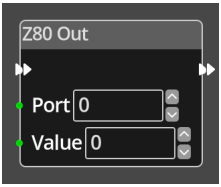
Jump/Fire

A constant integer value that will be output on the **Jump/Fire** port if the input check matches the mode for the jump/fire input.

Mode

Define the mode of operation between *Pressed*, *Released*, *Just Pressed* and *Just Released*. See above for an explanation of the difference between the modes.

Z80 Out



The *Z80 Out Node* is a very special purpose node that allows you to perform an output operation on the Z80 in the Spectrum. It should be used with great care, as incorrect use can crash the machine.

An example use is to change the border colour, setting the port to 254, and the value to a colour value between 0 and 7 will change the border colour.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Port

An integer input port used to provide the port number to output the value to.

Value

An integer input port used to provide the value to output to the given port.

Parameters

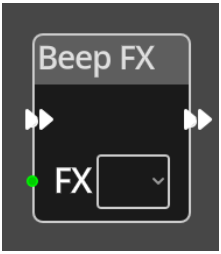
Port

A constant integer value for the port number, used when the **Port** port is not connected.

Value

A constant integer value for the value to output, used when the **Value** port is not connected.

Beep FX



The *Beep FX Node* is used to play a beeper sound effect defined in the Beep FX panel of the Sound Editor.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

FX

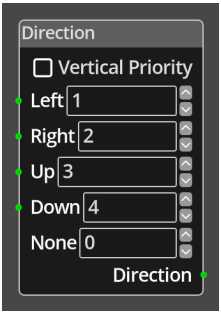
An integer input port used to provide the index of the Beep FX sound effect in the list as defined by the Sound Editor.

Parameters

FX

A dropdown selection of existing sound effects from the Beep FX tab of the Sound Editor. Use this if you wish to trigger a specific sound effect, use the port at the same level in the node if you wish to trigger a sound effect based on some calculations elsewhere in the logic graph.

Direction



The *Direction Node* is used to convert the typical four directional movement input into a single value representing the primary direction of movement. This is useful for things like switching sprite animation on Objects to show the correct animation for the movement. If the input suggests a diagonal movement, the output will be a single value representing the "dominant" axis as defined by the **Vertical Priority** parameter, if this is checked, then up/down will be chosen over left/right if diagonal movement is indicated, if it is not checked, left/right will get priority.

unlike other nodes with port/parameter pairs, in this node the direction rows are not port **or** parameter, both are required. The port is the input, usually from a Controller node, while the parameter represents the output value that will be used if that direction is determined to be the dominant direction. In the example here, if only the *right* controller is non-0, the **Direction** value output will be 2.

Ports

Left

An integer input port providing the input value for the left direction, any non-0 value will be considered movement in that direction.

Right

An integer input port providing the input value for the right direction, any non-0 value will be considered movement in that direction.

Up

An integer input port providing the input value for the up direction, any non-0 value will be considered movement in that direction.

Down

An integer input port providing the input value for the down direction, any non-0 value will be considered movement in that direction.

Direction

An integer output port, one of the four direction parameter values will be output on this port depending on the input data and the **Vertical Priority** parameter.

Parameters

Vertical Priority

A boolean value indicating which axis to give priority to if diagonal movement is indicated. If checked, up/down will get priority, otherwise left/right will.

Left

A constant integer value that will be output on the **Output** port if the inputs indicate movement left.

Right

A constant integer value that will be output on the **Output** port if the inputs indicate movement right.

Up

A constant integer value that will be output on the **Output** port if the inputs indicate movement up.

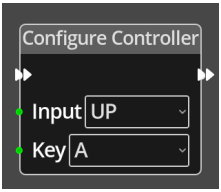
Down

A constant integer value that will be output on the **Output** port if the inputs indicate movement down.

None

A constant integer value that will be output on the **Output** port if none of the inputs indicate any movement.

Configure Controller



The *Contfigure Controller Node* sets the key mapping for a specific controller input to a new Keycode, thus allowing the game designer to offer the ability for a player to set their own keys for playing the game.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Input

The index of the controller input type to set the key mapping for: 0 - UP, 1 - DOWN, 2 - LEFT, 3 - RIGHT, 4 - FIRE. Key The Scancode of the key to assign to the controller input.

Parameters

Input

A dropdown list of the possible controller inputs to choose from in place of a connection at the **Input** port.

Key

A dropdown list of all the Scancodes to choose from in place of a connection at the **Key** port.

Wait No Key

The *Wait No Key Node* simply stops all logic processing until no keys are held down. This is used to avoid actions rolling over into other actions. For example if a check for a key switches a game element on if it wasn't already on, and the same key switches it off if it was, then waiting for the key to be released in between prevents the object rapidly switching between on and off.

Ports

Flow In

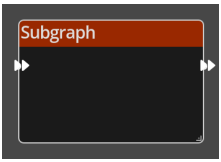
In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Subgraphs

Sub Graph



The *Subgraph Node* is a means to "call" a subgraph flow defined in the Shared Logic editor. The input and output ports on the *Sub Graph Node* will depend on the contents of the Subgraph it is calling.

See Shared Logic for more information regarding the use of Subgraphs.

Ports

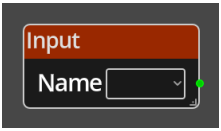
Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Input



The *Input* node represents an input value to a Sub Graph. The provided name is used on the sub graph node as the name for the input port that provides the value into the rest of the sub graph flow.

See Shared Logic for more information regarding the use of sub graphs.

Ports

Out

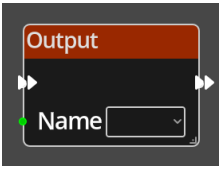
An output port, the value passed into the matching named port on the sub graph node will be provided here to the rest of the nodes in the sub graph.

Parameters

Name

The name of the port on the sub graph that will represent this input value.

Output



The *Output Node* represents an output value from a Sub Graph. The provided name is used on the sub graph node as the name for the output port.

It is entirely normal to use the same name on multiple output ports, only one will provide the output value to the port on exiting the sub graph, the last one encountered in the flow. This allows you to set the output to different values depending on the flow throught the sub graph.

See Shared Logic for more information regarding the use of sub graphs.

Ports

In

An input port, the value passed into this port will be provided up to the sub graph node on exit.

Parameters

Name

The name of the port on the sub graph that will represent this output value.

Flow In

The *Flow In Node* is only used in Sub Graphs. It marks the entrypoint for a flow contained in a sub graph. When a sub graph is used in another flow, this is the point that the execution continues when entering the sub graph node.

Ports

Flow Out

The flow for the sub graph will begin with the node connected to this port. The Flow In nodes acts in a similar manner to **Trigger** nodes, in that it has no input flow, as it defines the entrypoint to a sub graph flow.

Game

Reset Game

The *Reset Game Node* resets the game. This operation returns all objects to their initial position, runs the Initialise flows on all objects, locations, screens and the global logic, and resets all variables to their initial values. It will delete all dynamically spawned objects. It will reset the starting location, but not modify the current screen.

Ports

Flow In

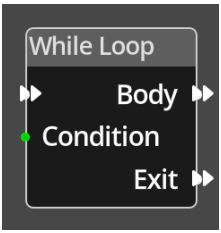
In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Control

While Loop



The *While Loop Node* will run the node flow that is connected to its *Body* output port repeatedly while the result of evaluating the input at the **Condition** port is non-zero.

When the loop is complete, the logic graph will continue with the flow connected to the *Exit* port.

Ports

Flow In

> In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Body

> The nodes that are to be repeatedly executed during the loop. Execution will continue while there is a node connected to the Flow Out of the last node executed, the body ends when there is no connection to the Flow Out, at which point the index is updated and the node will check if it is to run the loop again or exit.

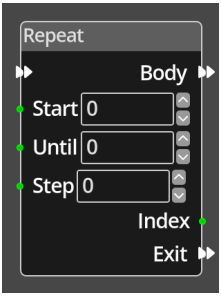
Condition

An integer input port providing the value to check at each loop iteration. If the value is 0, the loop exits, if it is non-zero, it loops again.

Exit

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Repeat



The *Repeat Node* will run the node flow that is connected to its *Body* output port repeatedly while an index is between the *Start* and *Until* values provided, altering the index value each time around the loop by the *Step* value.

The index is first set to the *Start* value, and then it is compared to the *Until* value, if the *Step* is positive and the index is greater than or equal to the *Until* value, the loop exits, similarly, if *Step* is negative and the index is less than or equal to the *Until* value, the loop exits. That is, the loop runs while the index is *before* the *Until* value, it is "exclusive".

During the looping, the current value of the index can be accessed by nodes in the *Body* using the *Index* port. Before execution, the index port is undefined, after execution it will contain the last value of the index before exiting the loop.

When the loop is complete, the logic graph will continue with the flow connected to the *Exit* port.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately

originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Body

The nodes that are to be repeatedly executed during the loop. Execution will continue while there is a node connected to the Flow Out of the last node executed, the body ends when there is no connection to the Flow Out, at which point the index is updated and the node will check if it is to run the loop again or exit.

Start

An integer port providing the start value of the index, this will be the value of the index during the first run of the *Body*, it is "inclusive".

Until

An integer port providing the value to check the index against for exiting the loop. The loop is exited before the index reaches this value.

Step

An integer port providing the value to add to the index each iteration through the loop. This value can be positive or negative, allowing loops where the index increases or decreases with each iteration.

Index

An integer output port that provides the current value of the index in the loop, typically used by nodes within the *Body* flow.

Exit

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

Parameters

Start

An integer input to specify a fixed value for the start of the loop index.

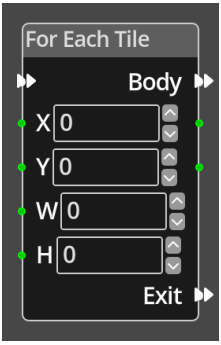
Until

An integer input to specify a fixed value for the exit point of the loop.

Step

An integer input to specify a fixed value to be added to the index each iteration through the loop.

For Each Tile



The *For Each Tile Node* is used to make changes efficiently across a rectangular area of the Tilemap.

Ports

Flow In

In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Body

A node connected to the **Body** port will be executed in sequence for every tile in the rectangular area.

X

An integer input port used to provide the X value of the top left tile coordinate.

Y

An integer input port used to provide the Y value of the top left tile coordinate.

X

An integer input port used to provide the width in tiles of the area to process.

H

An integer input port used to provide the height in tiles of the area to process.

X Out

An integer output port, used to provide the X tile coordinate to the **Body** flow, each iteration through the loop, this value is updated to represent the current tile in the area.

Y Out

An integer output port, used to provide the Y tile coordinate to the **Body** flow, each iteration through the loop, this value is updated to represent the current tile in the area.

Exit

A node connected to the **Exit** port will be executed in sequence following the completion of this node's operation.

Parameters

X

A constant integer value for the X coordinate, used when the **X** port is not connected.

Y

A constant integer value for the Y coordinate, used when the **Y** port is not connected.

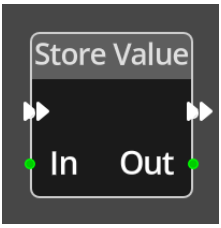
Collision

A constant value for the collision sides, used when the **Collision** port is

not connected.

General

Store Value



The *Store Value Node* takes a single input value and stores it for future use. When the **Out** value is requested, it returns the value that it stored, without running any nodes that connect to the **In** port as would normally happen.

This node is very useful to optimise the execution of logic. Under normal conditions, whenever a node requests the value on any of its input ports, the logic will be executed on the node that connects to that port, and any nodes that connect to that node, and so on. This can be expensive if the value is used multiple times, especially if it doesn't change. So putting a *Store Value Node* in place ensures that the calculation only happens once, and the resulting value is reused.

Ports

Flow In

> In order for this node to perform its operation, it must be connected into an active flow using this input port. The flow will ultimately originate at a **Trigger** node but can come from the **Flow Out** port of any other flow node.

Flow Out

A node connected to the **Flow Out** port will be executed in sequence following the completion of this node's operation.

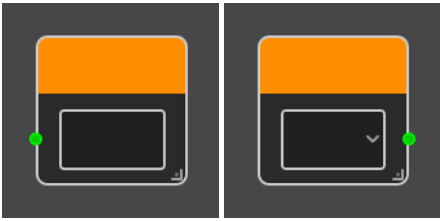
In

An integer input port used to provide the value to store for future use. This port will only request its value once, no matter how many times the **Out** port is used.

Out

An integer output port that provides the stored value.

Portals



The *Portal* nodes operate as pairs, they are only for the purposes of editing logic, they have no impact on the final code that Kwyll executes, neither in terms of memory use or performance. They provide a way to help keep your logic graphs a little less cluttered.

The *Portal In* node allows you to input any value, such as the result of an operation in the flow, you can give the *Portal In* node an arbitrary name. You can then create an *Portal Out* node, and select the name of the *Portal In* node you created from the dropdown provided. The value that is input to the matching *Portal In* node will be made available on the output port of the *Portal Out* node, as if the two nodes were connected by a wire. This allows you to use values elsewhere in your logic without having wires dragging all accross the graph making it difficult to follow.

It is possible to have multiple *Portal Out* nodes for each *Portal In* node, just as you can connect the output port of a node to the input port of many other nodes.

NOTE

> The same rules apply with regards to all data flow logic nodes, that is, if you connect the output of the *Portal Out* node to some other node, that node will "request" the value, which in turn will request it from the *Portal_In* node and it will request the value from the node connected to its input port. Just as if the end node were directly connected to the start node by a wire, which will result in the logic providing the input value to generate code on the export. Do not use *Portals* in place of Store Value nodes.

Ports

In

On the *Portal In* node, an input port, the value passed into portal that will appear at the output port of any matching *Portal Out* node.

Out

On the *Portal Out* node, an output port, the value on this node will be the value provided on the input of the matching *Portal In* node.

Parameters

Name

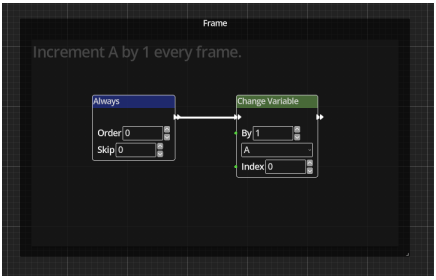
On the *Portal In* node, an arbitrary name for the portal, choose something that makes it easy to remember when you use the portal's value on a *Portal Out* node. On the *Portal Out*, this will be a dropdown containing a list of all defined *Portal In* nodes on the current logic graph.

Comment

The *Comment Node* is a non-operational node, it doesn't connect to any flow or have any input or output ports or parameters. It has a single multi-line text field that you can enter any text into. The text will be saved with the game data, but typically has no bearing on the exported data. This node is usually used to provide some useful information in more complex logic graphs, a description of the operation of a certain part of the graph, or instructions on how to use a subgraph etc.

It is resizable and will retain its size, allowing you to place it and size it to your requirements.

Frame



The *Frame* node is not a node as such, in that it provides no value to the actual logic, instead it offers a way to organise your logic and make it easier to keep track of what complex logic flows are doing.

When you create a *Frame* it will be placed into the grid as with all other node types, you can drag it around, and like some others including the Comment node, you can resize it. However, this is where the similarity to other nodes ends. The *Frame* node allows you to collect other nodes into a group. When you drag other nodes onto the *Frame*, or drag/resize the *Frame* to enclose other nodes, those nodes will be "captured" by the *Frame*. This does not affect how the other nodes work, it simply collects them into a group. Once the *Frame* has some nodes captured, when you move the *Frame*, the contents will move with it.

You can remove a node from a *Frame* by simply dragging the individual node outside the *Frame*.

In addition to simple grouping, the *Frame* node allows you to type arbitrary text in the background of the node. Simply click on the *Frame* background, and start typing. This is useful to describe the purpose of a group of nodes captured in a frame, much like the Comment node.

Tutorials

Simple Platformer Controller

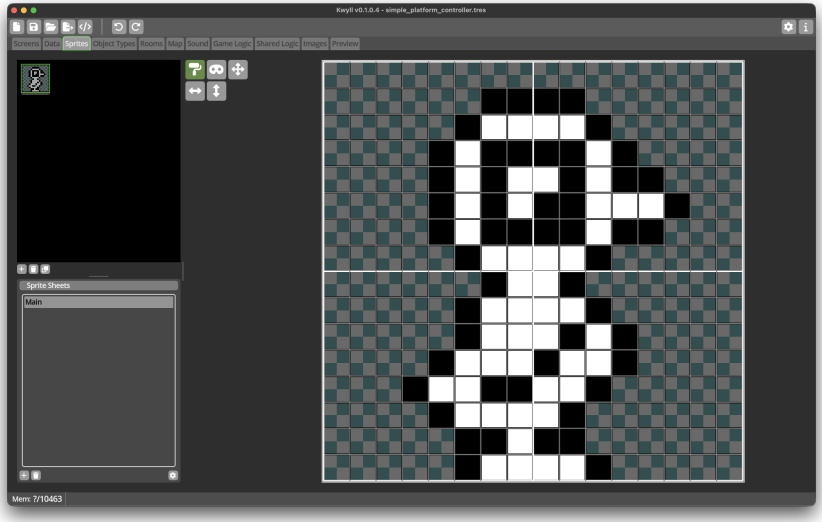
How To Create A Simple, Left, Right and Jump Platformer Controller

One of the most common game genres for retro computers is the platformer, a game where the player typically has controls to move left and right, and to jump onto platforms and fall down when walking off a platform. This tutorial guides you through creating the logic nodes to create a player object in this style. While this very likely won't be adequate for a fully fledged game it should give some insights into some of the techniques that Kwyll offers to do this and serve as a starting point for customisation and improvement.

Setup

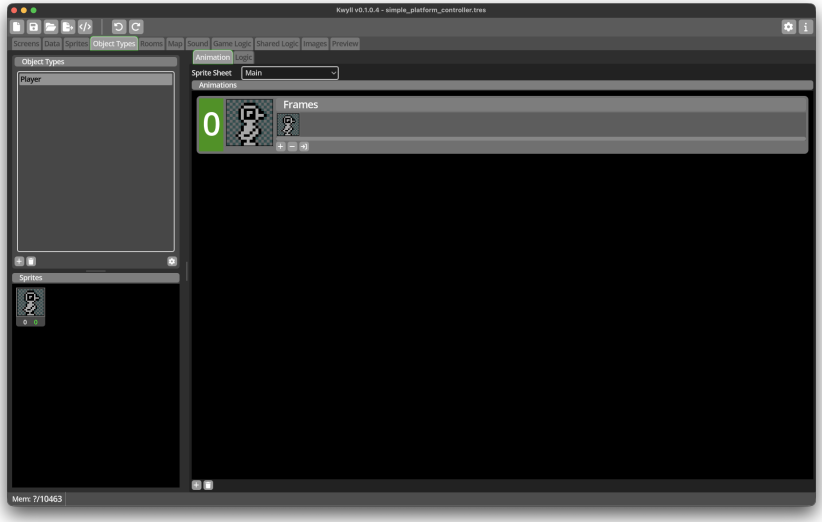
Sprite

First, lets start by creating a sprite for our character, something simple as we're not focused on animation for this tutorial.



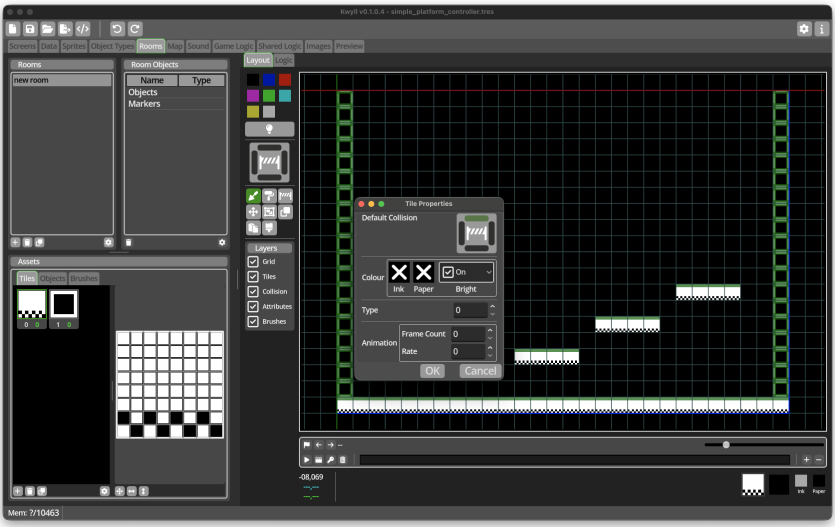
Object

Then we'll turn this sprite into a simple object, our Player, remember to name it appropriately and set the draw mode to mask if you've drawn a mask for your player sprite. Add a single animation with one frame and choose the sprite you created for your character.

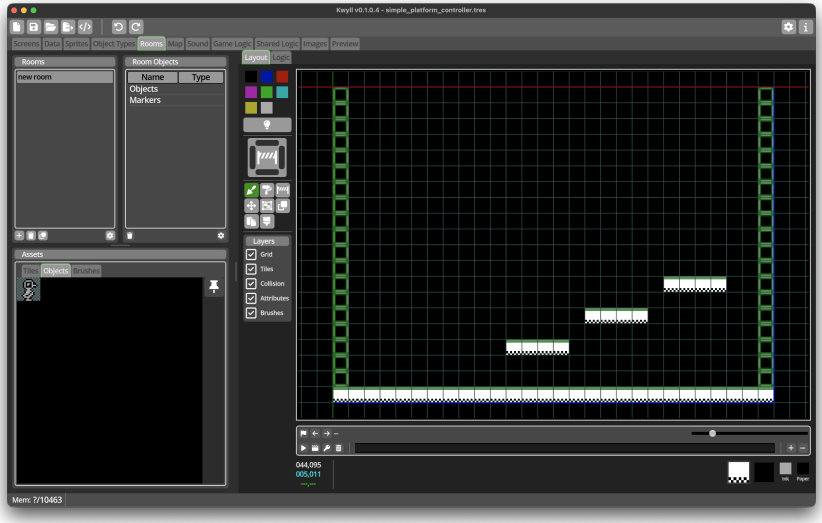


Room

Next we'll setup a simple room to test out the player. Create a single simple tile, in the properties dialog, make sure to set the default collision to "top" as this will be the tile for our platforms, we want to be able to jump up through it, but then land on the top and not fall through.

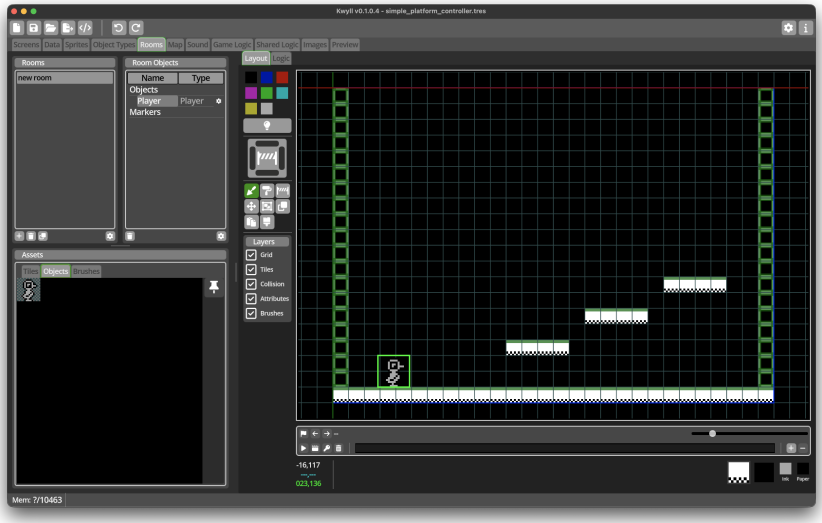


Draw a ground and some platforms for our player to jump on using the tile we just created.



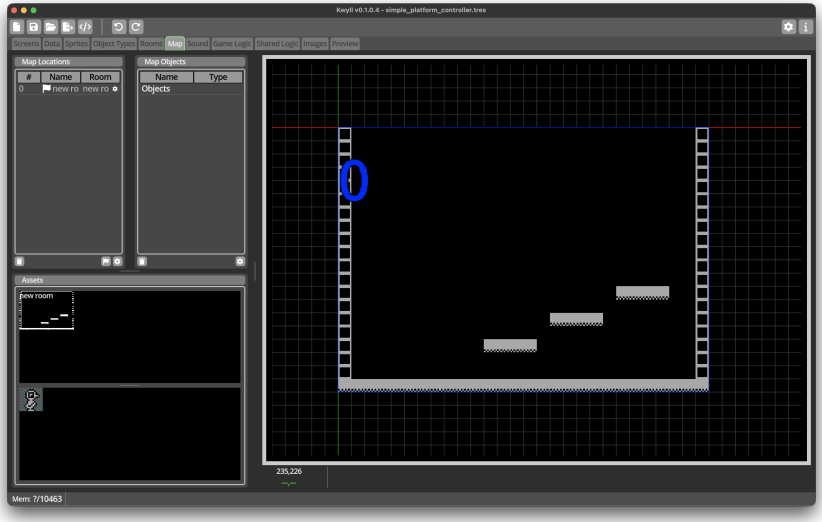
Drag the Player object from the objects panel onto the room. Note: normally the player would be a "Map" object, which allows it to travel

between rooms, but for the purposes of this tutorial, we'll just have a single room, so we can just place our player in the room directly.



Map

Finally, in order to be able to play the game, we must add the room to the map.



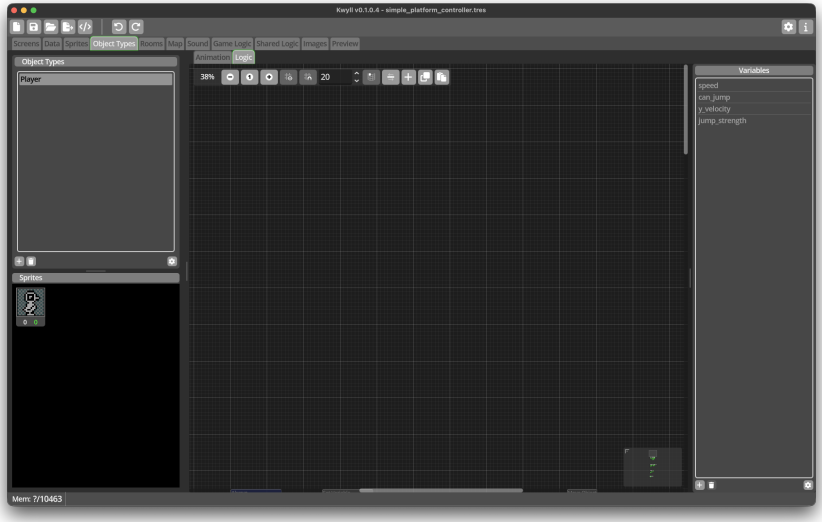
Logic

Now that we've got all the pieces in place, it's time to look at the core of this tutorial, the logic. In order to keep things clear, the different steps of the logic are implemented as separate flows, some of them could potentially be combined if you wanted to reduce the number of nodes, that is left as an exercise for the reader.

We break the logic down into 5 steps, movement, jump triggering, fall processing and landing.

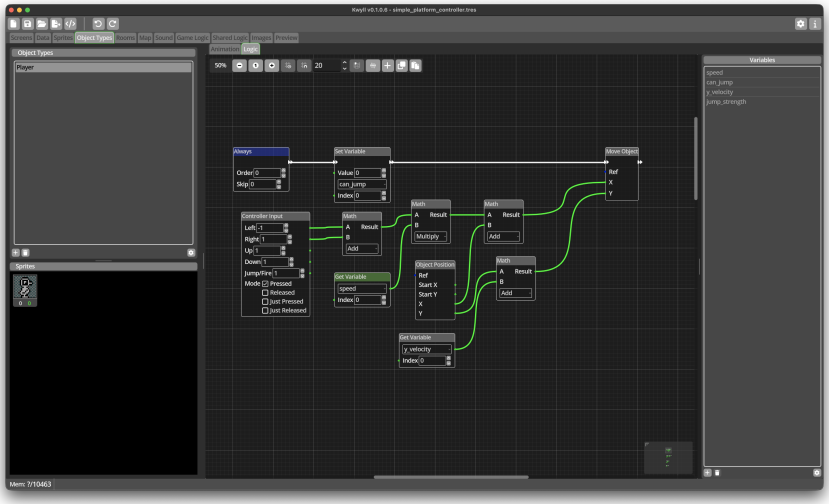
Firstly, we'll need some variables to use during the logic, add the following variables to the Player object logic:

- speed
- can_jump
- y_velocity
- jump_strength



Movement

This flow is triggered by an "Always" trigger so will run every frame. It serves two purposes, checking the left right keys and altering the player's position in the horizontal direction, and applying any jump/fall amounts to adjust the player's position in the vertical direction. Once this is done, it will attempt to move the player to the new location.



Following the flow connections (white) you can see the first thing that happens after the flow is triggered is to set the "can_jump" variable to 0, this is so that should the move cause the player to fall off a platform they won't be able to jump, if the move does not cause the player to start falling, this flag will be reset by the Landing flow described below.

The next thing in the flow is the actual move, this gets the X and Y position to move to from a set of calculations. Let's look at X first.

The "Controller Input" node is configured to return -1 for left and 1 for right, if either button is not pressed, it will return 0 for that direction. These two values are added together in a "Math" node, this is to cancel out should the player press both left and right simultaneously, as 1 + -1 is 0, if only one button is pressed, the result will be either -1 or 1 depending on whether left or right is pressed.

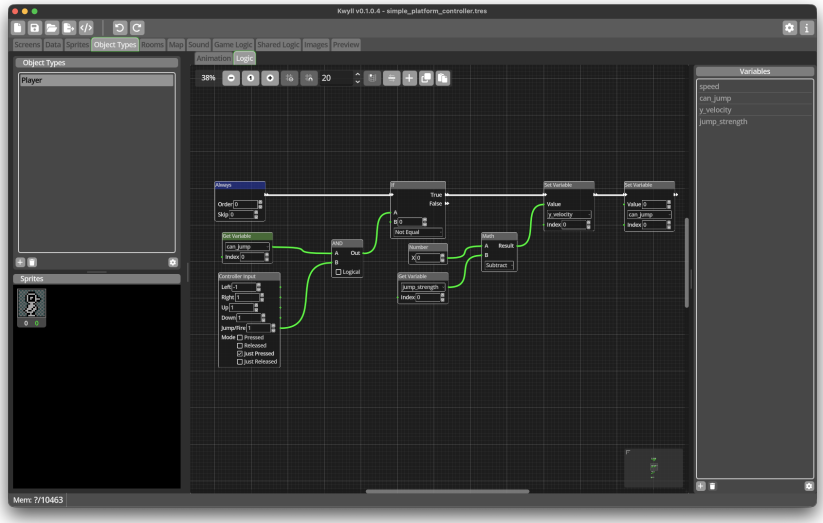
The result of this calculation is then fed into another "Math" node that multiplies the direction by the "speed" variable. The result of this calculation is then fed into a third "Math" node that adds it to the current X position of the player, moving it in the correct direction by the desired speed.

The Y position is calculated by adding the "y_velocity" variable to the current Y position of the player.

Finally, we try to move the player to the calculated new position.

Jump Triggering

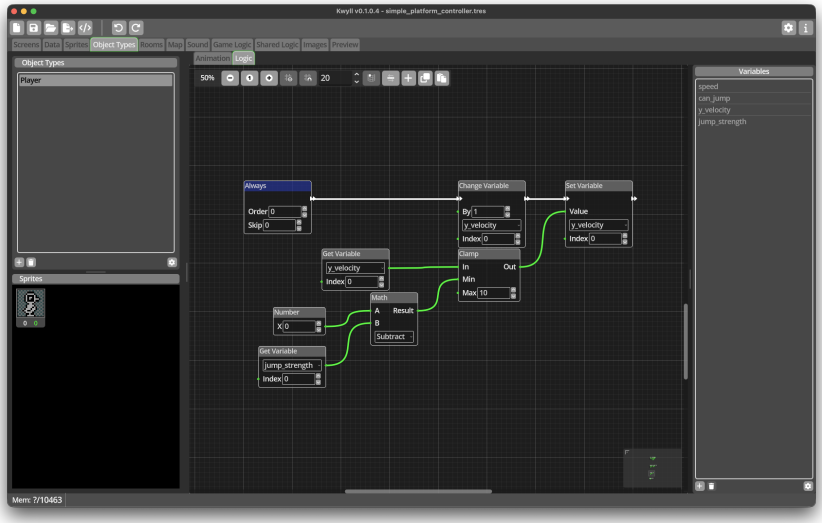
Next, in a separate "Always" flow, we check the jump button, note the "Controller Input" is set to "Just Pressed" for this, as we don't want the jump to be continuous as we do for the left/right movement, this is one good reason to keep the flows separate.



We get the result from the "Controller Input" to see if jump has just been pressed. We also get the value of the "can_jump" variable, and feed the result of both of these into an "AND" node. Only if the button is pressed, and the player can jump (both are 1), do we proceed to make the player jump. This is done by simply setting the "y_velocity" variable to minus the "jump_strength" variable, remember, negative in the Y direction is UP. We also set the "can_jump" to 0, this prevents the player from jumping again until they have landed.

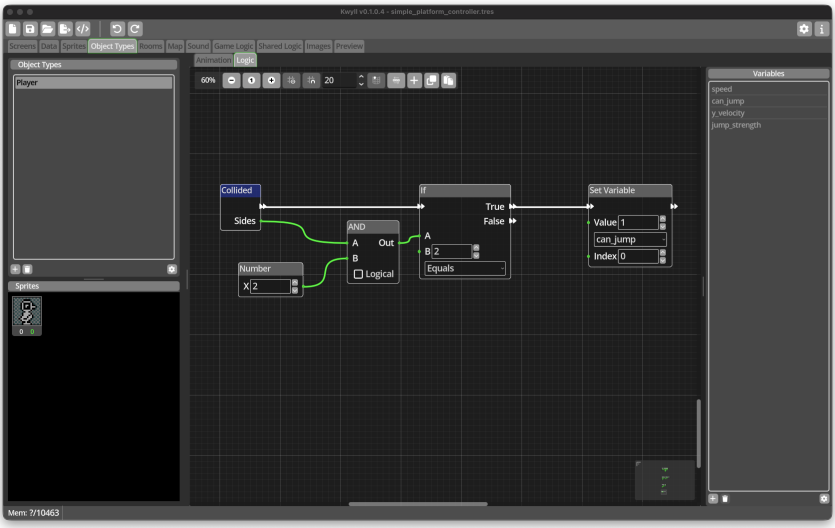
Fall Processing

Next, in yet another "Always" flow, we continuously adjust the "y_velocity" to account for gravity. We clamp the value to always be between minus the "jump_strength" and 10, this ensures the player never goes up by more than the defined jump strength, and never falls faster than 10 pixels per frame.



Landing

Finally for now, we have the logic flow that reacts when the player has collided with something. The "Collided" trigger is automatically run when the "Move Object" node in the Movement flow results in the player hitting something. For the purposes of this tutorial, we know it can only be a platform or the ground, but we check the direction for completeness anyway and to ensure that collision in other directions is cleanly handled as you continue to develop your game around this controller.



The flow first checks if the sides includes "TOP" which is 2, it does this by logically ANDing the sides value with 2. Recall that the sides is a bit field, bit 0 (value 1 if collided) is UP, i.e. the move resulted in the object hitting something from below, bit 1 (value 2 if collided) is DOWN, bit 2 (value 4 if collided) is LEFT and bit 3 (value 8 if collided) is RIGHT. The actual value of sides is a potential combination of these values, for example, if the object collided in both the DOWN and LEFT directions, the value will be 2 + 4 = 6, ANDing with 2, will remove any other directions from the value but the DOWN value, so if the player does collide both DOWN and LEFT, LEFT will be ignored, the output from 6 AND 2 is 2, as illustrated by the binary representation below.

```
           RLDU
6       00000110
AND
2       00000010
result  00000010
```

Only if both "bits" in the binary number are 1, will the bit be 1 in the result,

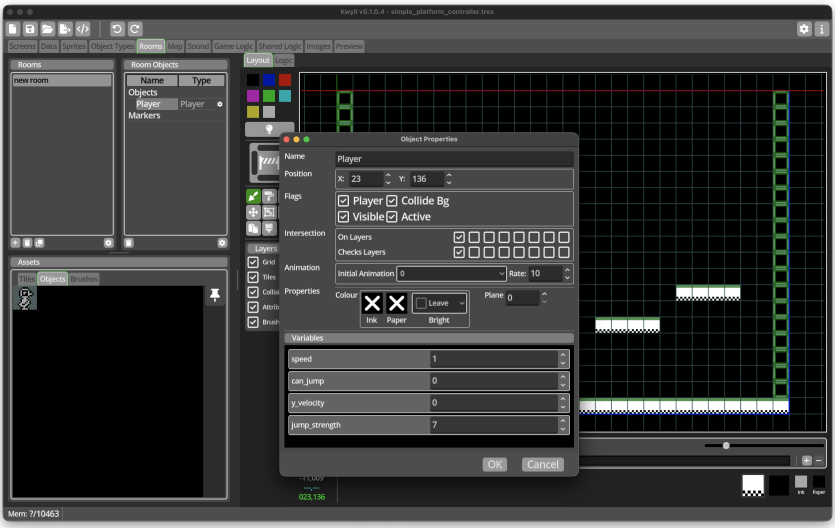
so the result of ANDing these two number will be 00000010 in binary, which is 2. This means we can just compare the result with 2 to check if the object collided in a downwards direction at all, irrespective of whether it collided in another direction as well or not.

If this is true, we set "can_jump" variable to 1 to indicate that the player can now jump again as they have landed.

Testing

Back in the Room Editor, open the object properties dialog for the player and make sure the "Collide Bg" flag is set so that the player object will collide with our platform tiles.

In the Variables section of the object properties, you'll need to set the values of "speed" and "jump_strength" to suitable values, 1 and 7 will be good for our example, but you can tweak those as you choose.



That should be all that is required to get a basic platformer control working in Kwyll. Switch to the Preview tab and start the game, you should be able to move left and right, and jump up onto the platforms. When you

walk off a platform the player should fall to the ground.