

MegaMan Engine for Unity2D

Made by MegaChibisX

Section 0: Unity

Page 2

Section 1: Player

Page 3

Section 2: Stages

Page 9

Section 3: Enemies

Page 12

Section 4: Menus

Page 14

Section 5: Building a Stage

Page 16

Section 6: Examples

Page 20

For any questions, you can contact me in the following places:

Twitter: <https://twitter.com/MegaChibisX>

DeviantArt: <https://www.deviantart.com/megachibisx>

Discord: MegaChibisX #0577

Section 0: Unity

Introduction

This tutorial assumes you have some basic experience with Unity. However, I have tried to explain things enough to allow people without Unity experience to, if not make a whole game with it, at least be able to make some stages with the already existing content. If you have any questions, feel free to ask @MegaChibisX on Twitter.

This engine was made with Unity 2018.2.18f1, and it should be compatible with later versions.

Packages

This engine has been made using as few packages possible.

2d-extras by Unity-Technologies is used as an expansion to Unity's default Tilemap. 2d-extras allows for things like **Animated Tiles** and **Rule Tiles**, or tiles that automatically find if they are a corner tile, an edge tile, etc. Source: <https://github.com/Unity-Technologies/2d-extras>

If you are using a version of Unity greater than 2018.3, you will need to update the packages. Packages for 2018.2 and 2018.3 have been included in the Project Folder, in "VersionPackages".

To update the packages from within the folders, use the following steps:

- Close Unity, if Unity is open.
- Navigate to the right folder.
- Copy the contents of the folder.
- Navigate to "Assets/_Packages".
- Delete everything in this folder.
- Paste the contents of the folder you copied in their place.
- Open Unity.

If you're using a newer version of Unity and newer versions of the packages have come out, you will need to do a few things first:

- Close Unity.
- Navigate to "VersionPackages" inside the Project folder, and create a new folder named after your current version of Unity.
- Go to the page of each package and download the current version.
- Extract each package inside the folder you have created.
- Follow the steps from above as normal.
- Now everything *hopefully* works as expected.

If you're using a newer version of Unity and one or more of the packages have gotten obsolete:

- Serenade help you.

Section 1: Player

In this engine you are able to create multiple players. Mega Man, Proto Man, Bass, Mega Man Jet, Mega Man Power and X come with this project by default. For instructions on making your own character, view

4_PlayerTutorial.

The MegaMan GameObject

- The Parent GameObject called “Mega Man”.

Components:

- A Rigidbody 2D component is required for Unity to handle the game physics.
- A Box Collider 2D is also present in the Parent GameObject. This can be placed with the rest of the colliders as an individual GameObject, but it doesn’t make a difference, as long as there aren’t two or more colliders in the same GameObject, and the GameObjects are active at all times.

- The most important script in the Parent GameObject is the Player script, which handles everything that is player related. In-depth explanation of how the script works follows this section.

- Sprite

- “Sprite” holds the player’s animations.

Components:

- The Sprite Renderer should be active only for debug purposes, or if the player doesn’t need to change their colors to match their weapons. The Sprite Renderer is required in this GameObject, even if it is going to be hidden, as the objects which hold the colors (see below) will use this as a reference.
- The Animator component is what holds the Player’s animations. Although it can be done through the Animator itself, deciding which animation should be played is done through script.

- Colors

- Holds all the color-changing sprites.

Components:

- The Misc_SnapToGrid script is disabled, as it makes the Player jittery on moving platforms. Otherwise, it would be used in this object to make the player look like they move pixel-by-pixel, while not snapping moving the Rigidbody and possibly messing with any calculations. It can safely be deleted if you wish so.

- LightColor, DarkColor, Outline and Face

- All of these objects function the same, so they can be grouped together.
- Each of these objects holds one of MegaMan’s weapon colors, with “Face” being the parts that shouldn’t be colored. Each object checks the inactive sprite of “Sprite”, looks for a blank sprite with the parts of the player that should be colored the required color with the same name, uses that sprite in its sprite

Components:

- Sprite Renderer renders the part of the sprite and tints it appropriately.
- Misc_CopySprite reads the sprite from “Sprite” and updates the Sprite Renderer Component.

- Colliders

- This object just holds all the collider objects in it.

- SlideCollider

- Similar to the main Collider, but shorter.

Components:

- A Box Collider 2D shorter than one block, enough for MegaMan to stand through.

- PlatformCollider

- This collider is used because Unity’s physics act a bit odd. A Box Collider moving against a grid will sometimes make the player stop moving while moving at a straight line.

Components:

- A Capsule Collider to give some roundness to the player’s collider so it doesn’t get stuck in the grid.

- MegaMan_StageSounds, MegaMan_WeaponSounds

- The first object plays audio related to MegaMan’s movements, like landing and getting hurt, and the second one plays audio related to MegaMan’s weapons, like charging and shooting.

Components:

- An Audio Source is required to play Audio in Unity, and each one can only play one audio clip at a time. Two objects should be enough to handle all of MegaMan’s audio.

- GearGauge

- This is the bar that shows how close MegaMan is to overheating when using his gears.

Components:

- The typical Mesh Filter and Mesh Renderer are present here, as the bar is basically a simple square with a sprite on it. The important thing is the shader. It's more technical than what needs to be known for this project.

- ExhaustSmoke

- The smoke MegaMan creates once overheated.

Components:

- The Particle System creates smoke in a semi-circle shape above MegaMan. Unity's Particle System has a lot of parts, so you will have to look into them yourself.

The Player Script

[AddComponentMenu("MegaMan/Allies/Player")]

- This line simply puts the Player script in the right category. It isn't necessary, but it does look nicer.

public static Player instance

- It is faster to have a static variable to keep track of the player, every time we need to access their script from another one, than to search for them with FindObjectOfType<Player>().

float timeScale = 1.0f

- As the player can be slowed down, and at a different rate from the enemies, this keeps track of how fast the player moves. For example, "timeScale = 0.5f" means the player moves at half speed.

float deltaTime, float fixedDeltaTime

- They act just like Time.deltaTime and Time.fixedDeltaTime, but use the Player's unique timeScale.

Animator anim

- The Player's Animator component, which is usually used to animate the Player. In this case, it just holds the animations, and which one is played gets decided by **protected void Animate()** (see below).

Rigidbody2D body

- The Player's Rigidbody component, in charge of handling the Player's physics.

SpriteRenderer sprite

- The Player's main sprite. This is usually hidden in-game, but it is used as a reference from the individual pieces of the player's sprite, which can be colored individually, used for weapon colors.

PlayerHealthbars bars

- This keeps track of the Player's health and the energy of all their weapons.

Pl_WeaponData.WeaponColors defaultColors

- As the player can be any color, keeping track of the default colors is important. The WeaponColors struct in Pl_WeaponData does exactly that, and it is used here to give the Player their colors when they aren't assigned by the weapon they have selected.

Vector2 input

- Keeps track of the current keyboard input.

Collider2D normalCol

- The player's normal collider.

Collider2D slideCol

- The player's sliding collider.

GameObject spriteContainer

- As each color of the Player is an individual sprite, putting them all in a single GameObject, or "Colors" in this case, is easier and faster than accessing each one individually. They can be turned off quickly when the

Player's sprite shouldn't be seen.

AudioSource audioWeapon

- This is the AudioSource that plays the Player's weapon-related audio.

AudioSource audioStage

- This is the AudioSource that plays the Player's stage-related audio.

SpriteRenderer bodyColorLight, bodyColorDark, bodyColorOutline

- These three SpriteRenderers show each show different part of the player's sprite, which can be colored differently based on the active weapon and charge level. Their color is decided in **void LateUpdate()**.

GameObject speedGearTrail

- A prefab that is created every few frames, if the Player is using the Speed Gear.

ParticleSystem gearSmoke

- The smoke that the Player makes if they overheat from Gear Usage.

GameObject gearBar, Material gearBarMaterial

- The bar that shows how much Gear energy the Player has used.

GameObject deathExplosion

- The prefab of explosion that gets created when the Player dies.

PlayerSFX SFXLibrary

- The sound effects the Player uses.

Pl_WeaponData currentWeapon

- The weapon that the Player currently uses.

List<Pl_WeaponData.Weapons> weaponList

- A list of all the weapons the player can use.

int currentWeaponIndex

- The currently active weapon element in the list.

Menu_Pause pauseMenu

- The menu that appears when the game is paused.

bool paused

- Checks if the game is paused.

bool useIntro

- Checks if the player should use their intro when they spawn. Needs to be true when a new stage starts or when the player spawns, but false if the player is changed from the menu.

Menu_Cutscene cutscene

- The currently playing cutscene.

bool canMove

- Checks if the Player should respond to input.

enum PlayerStates

The possible states the Player can be in.

- Normal: The normal state of the Player.

- Still: The Player can't move.

- Frozen: The Player has been frozen and can break free.

- Climb: The Player is on a ladder.

- Hurt: The Player is in a knockback phase.

- Fallen: The Player has gotten knocked off their feet.
- Paused: The game has been paused.

PlayerStates state

- The current state of the Player.

float health, float maxHealth

- The health and max health of the Player.

bool canBeHurt

- Checks if the Player can be damaged.

bool canAnimate

- Checks if the Player's animations should be handled by the normal script. When unique animations are played, like a slash or a combo, this should be false. Otherwise, it should be true to react to movement, damage and attacking.

float knockbackTime

- The time left for the Player to be knocked back.

float invisTime

- The time left that the Player won't be able to get damaged.

bool gearAvailable

- Checks in the GameManager if the player has a Double Gear item collected.

float gearGauge

- The remaining gear energy the Player has.

bool gearActive_Speed, bool gearActive_Power

- Checks if either gear is active. Both can be active at the same time.

bool gravityInverted

- Checks if the gravity is inverted for the Player.

float moveSpeed, float climbSpeed, float jumpForce

- The run and climb speed of the Player, and the force of their jump.

float slideTime

- Checks how much time the Player has to slide.

float chargeKeyHold

- Increases if the weapon button is held, returns to 0 if it released.

float shootTime

- How much time the Player has left to show their shoot animation. Overrides all other animations.

float throwTime

- How much time the Player has left to show their throw animation. Overrides all animations except the shooting one.

float gearTrailTime

- Time to until the next Speed Gear trail should spawn. Always decrements, but the trail only spawn if the Speed Gear is also active. Resets when it hits 0.

bool gearRecovery

- Checks if the Player has used the Gears too much and they have overheated.

float width, float height, Vector3 center

- Keeps track of the Player's attributes from their normal collider.

bool lastLookingLeft

- Checks if the last left/right input was left.

Vector3 right, Vector3 up

- The Player's local right and up Vectors.
- Right gives the direction where the Player is looking at.
- Up gives the direction going over the Player's head.

The methods are mostly explained with comments inside Player.cs, so this will be a quick explanation of some key points.

protected virtual void Start()

- This method is called every time a new Player object is created, including when the stage starts.

protected virtual void Update()

- This method is called once every frame.

protected virtual void LateUpdate()

- This method is also called once every frame, but later than Update().

protected virtual void FixedUpdate()

- This method is called once every physics update, which happens after an almost set number of milliseconds.

protected virtual void OnDrawGizmosSelected()

- Draws stuff on screen when the object is selected.

protected virtual void OnGUI()

- Draws stuff on screen while the game is running.

protected virtual void OnCollisionEnter2D(Collision2D)**protected virtual void OnCollisionStay2D(Collision2D)****protected virtual void OnCollisionExit2D(Collision2D)****protected virtual void OnTriggerEnter2D(Collision2D)****protected virtual void OnTriggerStay2D(Collision2D)****protected virtual void OnTriggerExit2D(Collision2D)**

- Collision-related functions

protected virtual void HandleInput_Movement()

- Handles input related to the player's movement.

protected virtual void HandleInput_Attacking()

- Handled input related to the player's attacks.

protected virtual void HandleInput_Technical()

- Handles input for miscellaneous purposes, like the pause menu.

protected virtual void HandleInput_Timers()

- Keeps track of all the timers, and takes whatever action needs to be taken when a timer expires.

protected virtual void HandlePhysics_Movement()

- Handles physics related to the player's movement.

protected void Climb()

- Handles physics when the player is climbing.

protected void Animate()

- Plays the correct animation.

protected void KnockBack()

- Applies knockback when applicable.

protected virtual void Land()

- Handles actions when the player touches the ground.

public virtual void Damage(float damage)

- Damages the player.

public virtual void Kill()

- Kills the player.

public virtual void SetWeapon(int weaponIndex)

- Sets the player a weapon based on the weapons they have available.

public void RefreshWeaponList()

- Updates the player's available weapons.

public void SetGear(bool speedGear, bool powerGear)

- Sets each gear to enabled or disabled.

protected void MakeTrail()

- Makes a speed gear trail.

public void SetState(PlayerStates newState)

- Sets the current state the player is in, while updating some necessary variables based on each state.

public void CanMove(bool _canMove)

- Allows or prevents the player from responding to input.

public void SetLocalTimeScale(float _timeScale)

- Sets the player's local Time Scale, which speeds up or slows down the player.

public void SetGravity(float magnitude, bool inverted)

- Sets the player's local gravity.

public virtual void UpdateGameManager()

- Updates variables in the game's Game Manager.

public bool isInSand, public bool isInWater, public bool isGrounded.

- Checks if the player is in sand, in water or touching the ground.

public bool facesWall

- Checks if the player is facing a wall.

public bool canStand

- Checks if the player can fully stand up, which can be false if the player is sliding or ducking

The Camera GameObject

The Camera Prefab can be found under "Assets/Prefabs/Player".

Components:

- Camera Ctrl:

The Camera Ctrl script handles everything movement related about the Camera.

When selected as a GameObject, a blue rectangle will be shown in the Scene view. This rectangle shows the area of the stage that will be visible once the game starts. For a more in-depth explanation, look into the **Border** Prefab in the **Stages Section**.

Section 2: Stages

A typical MegaMan game includes a variety of stage elements, like spikes, water, moving platforms, items and disappearing(yoku) blocks. This engine includes a decent amount of these elements.

General Grid

As many stage elements are tile-based, they will be using the grid. For that reason, it is suggested that a grid is always added to a new stage.

An empty grid prefab can be found under “Assets/Prefabs/Environment”. Drag the prefab named “Grid” into the Scene view.

The Tile Palette window also needs to be open so you can edit the grid. If there isn’t a Tile Palette in your Unity window, it can be found on the top left corner of Unity, under “Window/2D/Tile Palette”. Place the Tile Palette window somewhere, preferably not in the same area as the Scene view.

Each Active Tilemap layer acts as a separate stage element. To select the right element, click on Active Tilemap in the Tile Palette, and click on the desired item.



Ground

The Ground layer is the most basic layer in each stage. It allows the player to stand on it.

As a GameObject, it will need a **Tilemap Collider 2D** component and be assigned a **Solid (8)** Layer.

Water

The Water layer gives the player some increased jump height, a lower descend, and lets some water-based enemies act at the peak of their abilities.

As a GameObject, it will need a **Tilemap Collider 2D** component marked as Trigger and be assigned a **Water (4)** Layer.

Spikes

The Spikes layer instantly kills the player. Do not use any tiles other than the ones with spikes, as invisible spikes are very rude.

As a GameObject, it will need a **Tilemap Collider 2D** component, a **Rigidbody 2D** with a Static Body Type, as well as an **Enemy** script, with “Can Be Hit”, “Shielded” and “Destroy Outside Camera” all false. A **Solid (8)** Layer should also be given, so the spikes act as a ground when the player is invincible.

Ladders

The Ladders layer allows the player to go up and down on ladders. It doesn’t contain any mystical powers, it’s just a ladder.

As a GameObject, it will need a **Tilemap Collider 2D** component marked as Trigger, and also be assigned a **Ladder (10)** Layer.

Ice

The Ice layer makes solid ground slippery for the player. Ice tiles need to be placed **on top** of ground tiles.

As a GameObject, it will need a **Tilemap Collider 2D** component and an **Ice (11)** Layer.

Sand

The Sand layer makes the player slowly sink, instead of straight falling into the ground.

As a GameObject, it will need a **Tilemap Collider 2D** marked as Trigger, a **Stage_Sand**

component, as well as be assigned a Sand (12) Layer.

ConveyorLeft & ConveyorRight

The Conveyor layers move objects left and right at a fixed speed.

As a GameObject, it will need a **Tilemap Collider 2D** component and a **Solid** (8) Layer. A tag will also need to be given to the GameObject, with **ConveyorRight** and **ConveyorLeft** moving the objects accordingly.

GravityUp & GravityDown

Found in “Assets/Prefabs/Environment”, these two objects will switch the player’s gravity to face up or down, based on the direction the arrow shows. Just place them in the stage and touch them with the player.

LadderTop

Because I suck, as MegaChibisX, if this hasn’t been fixed until the engine releases, all the ladders in the grid won’t allow the player to stand on them once on top. For that reason, a semi-solid platform needs to be used on top of every ladder, to allow the player to move up but not fall down.

A **Platform Effector 2D** component is used to achieve the semi-solid platform behavior.

Yoku_Controller

Yoku, or disappearing, blocks can be a complicated bunch. The Yoku Controller will control a group of Yoku blocks, setting the order which they appear in, the time it takes for the yoku blocks to disappear and the sound that plays every time new yoku blocks appear.

A Yoku Controller only controls the speed the Yoku Block appear and disappear. Each Yoku Block needs to be its own GameObject. and it will need a **Stage_YokuBlock** component, as well as a **Box Collider 2D** component.


The Yoku Controller can activate multiple Yoku Blocks at once, as long as they are parented together. If multiple blocks are parented together, they will need a **Rigidbody 2D** component with a Static Body Type, given to the parent of the Yoku Blocks. If only one block gets activated at a time, it can have the **Rigidbody 2D** component by itself. It is still recommended that it is placed inside a parent GameObject.

Checkpoint

Checkpoints are necessary for a game as hard as a MegaMan game.

As soon as a player touches a checkpoint, the position of the Checkpoint GameObject will become the position the player spawns on the next time they die.

A Checkpoint GameObject will need a **Stage_Checkpoint** component, a **Rigidbody 2D** and a **Box Collider 2D** component. The **Box Collider 2D** will need to be marked as Trigger, and it needs to be big enough for the player to touch. You can edit the collider from the Scene view by pressing the

“Edit Collider” button 

Border

In almost all games, a MegaMan stage is split into many individual rooms. The player usually enters one room after another, until they reach a boss.

In this engine, switch between rooms is achieved with the **Stage_BossDoor** script. However, this script depends on the existence of another script, the **Stage_ChangeCamera** script.

A “Border” prefab can be found in under “Assets/Prefabs/Environment”, or a **Stage_ChangeCamera** script can be added to an empty GameObject.

When a GameObject with a **Stage_ChangeCamera** script is selected, a blue rectangle will be shown in the Scene view. That rectangle shows the Camera's possible Viewport, if the Camera is using these Borders in this room. Change the "Left Center", "Max Right Movement" and "Max Up Movement" variables to change the position and size of the rectangle.

To set the first Camera Borders in the beginning of a Stage, select the Camera from the Hierarchy window, find the **CameraCtrl** script in the Inspector and drag the Camera Borders GameObject from the Hierarchy window to the "Start Borders" field in the **CameraCtrl** script.

Transition_Empty & Transition_Door

Transitions allow the player to switch between Camera borders. **Transition_Empty** and **Transition_Door** both use the **Stage_BossDoor** script.

A transition GameObject needs to be placed between two Camera Borders, and each Border GameObject needs to be dragged from the Hierarchy window into either the "Borders Red Side" or "Borders Green Side" fields. When selected, a red sphere and a green one will appear in the Scene view, matching with the "Borders Red Side" and "Borders Green Side" fields. The Camera Border closer to the Red Sphere needs to be placed in the Red Side, and the same goes for the Green Side.

Section 3: Enemies

All the enemy scripts need to inherit from the script simply called **Enemy**.

The Enemy Script

All enemies require a **Rigidbody 2D** component to deal with collisions.

Each enemy script contains variables to keep track of the enemy's **health**, its contact **damage**, if the enemy **can be hit**, if the enemy is **shielded**, if the enemy should be destroyed when they aren't in view (**destroyOutsideCamera**), as well as the explosion it will create once it gets destroyed.

The enemy script also contains a few basic methods, which will usually contain the basic actions an enemy will need to have. They are best explained as comments in the **Enemy.cs** file itself.

protected virtual void Start()

This method is always called when the enemy is created, which is either when the stage starts, or when the enemy enters the Camera view.

protected virtual void OnDrawGizmosSelected()

This method shows stuff into the Scene view when the GameObject is selected. In this case, a cyan sphere is shown at the given center of the enemy.

protected virtual void Damage(Pl_Weapon weapon)

Damages the enemy based on weapon they are hit by.

protected virtual void Damage(float dmg, bool ignoreInvis)

Damages the enemy, and checks if their invisibility frames should be considered.

protected virtual void Kill(bool makeItem)

Kills the enemy, and prevents item drops from dropping if desired.

protected virtual void Despawn()

Despawns the enemy, without triggering any death behaviors.

public virtual void Shoot(Vector2 shootPosition, Vector2 shotDirection, float shotDamage)

Shoots a basic enemy shot at a position **shootPosition**, moving in the direction **shotDirection** and dealing **shotDamage** damage.

public virtual void LookAtPlayer()

Looks at the player.

public virtual Player GetClosestPlayer()

Returns the closest player.

The Enemy Blueprint Script

Enemies placed in the Scene view by themselves will be created as soon as the stage starts, and they won't respawn after they've died or despawned. To make enemies spawnable, they need to be saved as a prefab, and be assigned to an **Enemy Blueprint** component.

Create a new GameObject by right clicking inside the Hierarchy window and pressing "Create Empty". Go to "Add Component" and navigate to "MegaMan/Enemy/_Blueprint". Now drag the desired enemy from the Project window into the "Blueprint" field and press the "Recalculate Bounds" button.

Once the Blueprint is set up, a sprite of the enemy assigned to the Blueprint will be shown, and a cyan rectangle will surround it. The rectangle indicates the area which, if the player is in, will get the enemy to spawn.

Enemies



Metall



Neo Metall



Metall DX



Metall EX



Metall Swim



Space Metall



Metall Mommy



Metall Babies



Metall K1000



Metall Cannon



Octopus Battery



Super Ball Machine Jr.

...and more, probably.

Section 4: Menus

Menus in games can be used in many different ways. In this game, menus can be anything that displays something on the screen and stops the progress of the game while it is open. Pause menus, cutscenes and even boss select screens can all use the same basic **Menu** script.

public virtual void Start()

This method is called as soon as the menu is called.

public virtual void Update()

This method is called once every frame.

public virtual void Exit()

This method is called once the menu stops closes.

public virtual void DrawGUI()

This method draws stuff on the screen.

A few variables are used in this method, which need to be used to set the right position of everything in the screen.

cmrBase is the position inside the window where the right black border ends and the game viewport starts.

blockSize keeps track of the size of one block (or 16 in-game pixels) in the screen.

GUI.contentColor changes the color of the text shown in the menu.

Fundamentally a menu is simple. All you need is to define a the menu in a **Monobehaviour** script, like **public Menu menu** for example.

The complexity of a menu comes from what the developer needs it to do. The **Pause Menu** that comes with this project sure is a doozy. Same goes for the **Stage Select Menu**.

The **Pause Menu** is split into 5 sections, each dedicated to one aspect of the player.

Each section has an **index**, which keeps track of which item is currently highlighted in the section.

The section that is currently active is called **public SelectModes mode**.

The **Players Section** is the section on the very top.

Pressing the left and right buttons switches between players.

Pressing the jump button starts the game and switches to the currently selected player. Switching to a player like that, the player won't don a full intro.

Pressing the down button makes the **Weapons Section** active.

The **Weapons Section** allows the currently selected **Player** to select a weapon from what they have available.

Pressing the left/right/up/down buttons will move the selection to a different weapon in the list.

Pressing the jump button starts the game and switches the player's currently selected weapon to the selected one.

If up is pressed and the selection is on top of the list, then the **Player Section** will become active.

If down is pressed and the selection is on the bottom of the list, then either the **Recovery Items Section** or the **Utility Items Section** will be selected, depending on the horizontal position of the selection.

The **Recovery Items Section** contains items that have recover the player's health or weapon energy.

Pressing the left and right buttons will move the selection between the possible items.

Pressing the jump button will use the selected item, without starting the game.

Pressing the jump button when the last item in the list is selected will switch to the **Utility Items Section**.

Pressing the up button will switch to the **Weapons Section**.

Pressing the down button will switch to the **Lower Text Section**.

The **Utility Items Section** contains items that spawn some character or have some action in the game when selected.

Pressing the left and right buttons will move the selection between the possible items.

Pressing the jump button starts the game and spawns the desired character near the player, depending on the selected item.

Pressing the left button while the first item in the list is selected will switch to the **Recovery Items Section**.

Pressing the up button will switch to the **Weapons Section**.
Pressing the down button will switch to the **Lower Text Section**.
[This was never finalized]

The **Lower Text Section**, which on hindsight should have been named the **Misc Section**, contains everything else that doesn't fit the previous categories.

Pressing the left and right buttons will switch between the currently selected item.

Pressing the jump button will likely open another menu. Each item handles what action is taken when it is selected differently.

Pressing the up button will switch to either the **Recovery Items Section** Pressing the up button will switch to the **Weapons Section**.

Pressing the down button will switch to the **Lower Text Section** or the **Utility Items Section**.

Section 5: Building a Stage

The tutorial **1_StageTutorial** goes in more depth explaining this. This will give you a basic idea, but I recommend you just go to the tutorial.

When you create a new scene, there are a few things you need to put in it first. Here is a short tutorial on how to create a scene with some very basic elements.

First of all, you need to create a new Scene.

To create a new Scene in Unity, go to “File/New Scene”, or press Ctrl+N..

Delete all existing objects.

Normally a new Unity Scene will come with a few GameObjects, like a Camera and possibly a Directional Light. Select all objects and delete them.

Grab a Tilemap prefab.

You can make a Tilemap on your own by Right Clicking inside the Hierarchy window and going to “2D Object/Tilemap”. **However**, you can grab a premade Tilemap with everything you are going to need.

Go to “Assets/Prefabs/Environment” and drag the **Grid** prefab into the Scene view.

This Grid object has specific layers for Ice blocks (**Ice**), regular ground blocks (**Ground**), spike tiles (**Spikes**), water tiles (**Water**), ladder tiles (**Ladders**), sand tiles (**Sand**), conveyors moving to the left (**ConveyorLeft**) and conveyors moving to the right (**ConveyorRight**).

Grab a Camera prefab.

Go to “Assets/Prefabs/Player” and drag the “Main Camera” into the Scene view.

Grab the Player prefab.


Some prefabs, like this one, need to be in the Resources folder, so the players are there. Go to “Assets/Resources/Prefabs/Players” and drag your favorite player into the Scene view.

Start placing blocks.

Go to the “**Tile Palette**” window. If it isn’t open, go to “**Window/2D/Tile Palette**” and drag the window somewhere, preferably not in the same place as the Scene view.

Click the Brush  button.

Make **absolutely certain** that you have collected “Ground” as your active layer

Active Tilemap 

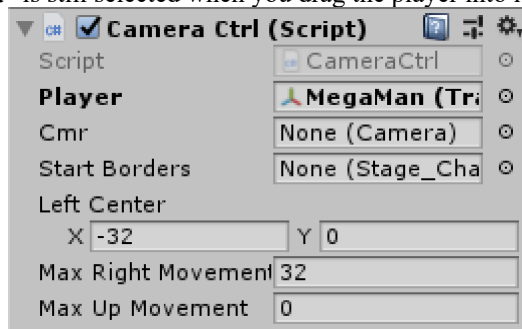
Pick any block from the Palette below.

Keep in mind that, while you can place each block on your own, the lower pitch black block will find and show the right blocks automatically



Tell the Camera to follow the Player.

Go to the “Main Camera” object and find the “Camera Ctrl” script. Click and hold on the “MegaMan” object, and drag it towards the “Player” field on the “Camera Ctrl” script. Make sure the “Main Camera” is still selected when you drag the player into it.



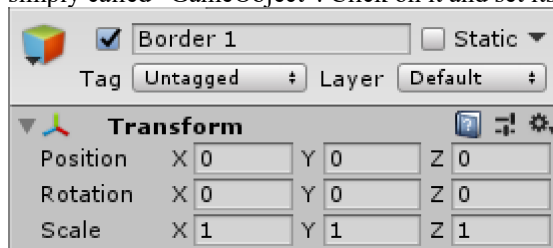
Test the room.

While not necessary, it is a good idea to test the room, to make sure everything has been done correctly. If you have done everything correctly, then MegaMan should be able to move around the room.

You will notice that the camera will only stay within the blue square that is displayed when the camera is selected. You can change the camera’s position and movement by editing the “Left Center”, “Max Right Movement” and “Max Up Movement” variables. This is however not recommended. Instead, we will set camera borders manually.

Set up Camera Borders.

Right click on the Hierarchy window and click on “Create Empty”. A new GameObject will be created, simply called “GameObject”. Click on it and set its position to (0,0,0). Name it “Border 1” too.



Click on “Add Component”, go to “MegaMan/Stage/Change Camera Borders”. You will see three variables called “Left Center”, “Max Right Movement” and “Max Up Movement”, as well as a blue rectangle in the Scene view. Set those variables until you are happy with the camera.

Duplicate “Border 1” by right clicking on it in the Hierarchy view and pressing “Duplicate”, or by simply pressing Ctrl+D. This will create another object called “Border 1 (1)”, which is stupid, so change it to “Border 2”.

Move the “Left Center”’s X so that the left edge of the rectangle will overlap with the right edge of the previous border. You can select both borders, by holding the Shift button and clicking on them, to view both rectangles at once. Additionally, with “Border 2” selected, you can click the little lock button on the top right corner of the Inspector View, and keep the object active in this view while another one is selected from the Hierarchy window. Click on the little lock again when you are done.

Duplicate the Border 2 again and do the same thing, calling the new GameObject “Border 3” instead “Border 2 (1)”, because once again this is stupid.

You should now have three rectangles next to each other, with 1 before 2 and 2 before 3.

Fill move of the stage.

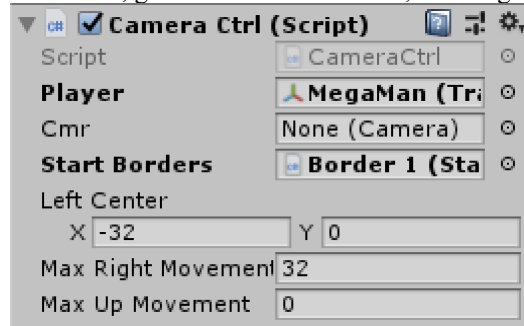
Add some more blocks if one of more of the rectangles don’t have a solid platform in them.

Add some Camera transitions.

The player will start with the Camera being inside “Border 1”’s borders, will continue to “Border 2”’s area,

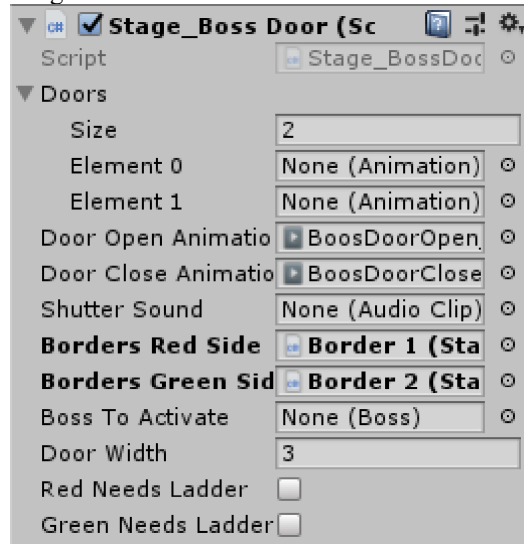
and end up in the “Borders 3” area, where they will face a boss.

First of all, go to the “Main Camera”, and drag “Border 1” into its “Start Borders” field.



Next, go to “Assets/Prefabs/Environment” and grab a “Transition_Empty” prefab. Drag it into the Scene and place it right between the rectangles of “Border 1” and “Border 2”. It will have its own light green rectangle, which shows Unity’s Box Collider 2D component. Scale the object upwards so it will cover the entire area the Player can go through between the two borders.

Drag “Border 1” into the “Border Red Side” field and “Border 2” into the “Border Green Side” field.



Try the stage again.

If everything has been done correctly, touching the edge of the screen should start a transition into the next room.

Add a boss door.

Go to “Assets/Prefabs/Environment” and grab a “Transition_Door” prefab. Drag it into the Scene view. Similarly to before, place it between the rectangles of “Border 2” and “Border 3”. Now drag “Border 2” into “Borders Red Side” and “Border 3” into “Borders Green Side”.

Try the stage again.

If everything has been done correctly, you should be able to slowly transition with the boss door as well.

Place some enemies.

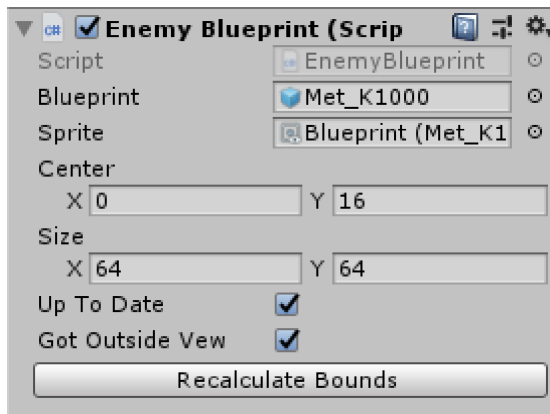
Go to “Assets/Prefabs/Enemies” and grab any of the enemies you fancy. Drag them into the scene, and place them wherever you fancy. You can run the game and try them.

Place some respawnable enemies.

Dragging enemies into the screen is fun, but for an enemy to respawn, it needs to be used as a blueprint.

Right click on the Hierarchy view, and click “Create Empty”. Now go to “Add Component”, then navigate to “MegaMan/Enemy/_Blueprint”.

Drag any enemy from “Assets/Prefabs/Enemies” into the “Blueprint”. Now click on the “Recalculate Bounds” button. The blueprint should now show the sprite of the enemy it contains, and the name should be something like “Blueprint (Enemy type)”.



Duplicate and place the blueprints at the desired spots inside the borders of “Border 1” and “Border 2”. Leave “Border 3” empty. You can run the game and try them again.

Place a boss.

Go to “Assets/Prefabs/Bosses/PharaohMan” and grab “Boss_PharaohMan”. Drag Pharaoh Man to the right side of the “Border 3” area.

Expand the “Boss_PharaohMan” GameObject in the Hierarchy window by clicking the arrow at the left of his name. Select the “LeftCorner” object and put it near the lower left corner of the room. Similarly, put the “RightCorner” object to the lower right corner of the room.

Select the “Transition_Door” and drag the “PharaohMan” object from inside “Boss_PharaohMan” into the “Boss To Activate” field.

Try the game out.

If everything has been done correctly, you should be able to enter the room and start your fight with Pharaoh Man now. Beat him and steal his weapon! Once Pharaoh Man is beaten, you should be able to exit the room again.

Section 6: Examples

A small game has been made to accompany this open source project. It's actually the project itself. It includes cutscenes, 8 Robot Master and their stages, 3 fortress bosses (one with 2 forms), 11 weapons, 2 Rush Utilities and a few enemies. You can edit these items or delete them and make your own. The project should provide plenty examples to understand how everything works, but the basics have their own guide found in the same folder as this.