Blind Guide Redesign Patterns

**Students:**

Victor Fortea Martinez, 500747258

Wasili Prattis, 500689732

**Chosen project:**

Blind Guide - 2D slidescroller game made using the Unity3D engine and the C# programming language. The game was created by Wasili Prattis and his Project Game Development (second year HvA project) team.

The player plays the role of a ghostly guide dog that wants to help his owner get to his home. There are multiple objects along the way, which can all be cleared using the player’s special powers.

The blind guy can only take one hit during the entire game; taking a hit means game over.

All obstacles and enemies can die by one or multiple powers and some obstacles require a combination sequence of multiple attacks in order to be stopped

Contents

[Singleton 3](#_Toc452897326)

[Flyweight 4](#_Toc452897327)

[State 5](#_Toc452897328)

[Structure 5](#_Toc452897329)

[Implementation 6](#_Toc452897330)

[Strategy 9](#_Toc452897331)

[Prototype 10](#_Toc452897332)

[Mediator 11](#_Toc452897333)

[Decorator 12](#_Toc452897334)

[Façade 13](#_Toc452897335)

[Observer 14](#_Toc452897336)

[Null Object 15](#_Toc452897337)

# Singleton

# Flyweight

# State

In Blind Guide, the player can choose between three different abilities to use in order to destroy obstacles and enemies. Seeing as only one of these powers can be active at a time, it is required that the player keeps track of the current power in order to decide which power to activate when the player presses the ‘Use Power’ button.

We accomplish this behaviour by using the state pattern: we create an array of available powers and switch between these powers whenever the player presses the ‘Switch Power’ button. By having an integer that tells us which power we’re currently using, we can activate the correct power whenever the player attacks.

The first power the player can use is telekinesis. The player can pull certain objects, that are marked appropriately with a tag, towards him and then throw them away to prevent the Blind Guy from being harmed.

The second and third power the player can use are quite similar; the player can breathe fire or ice in order to burn or freeze objects. Seeing as these powers are quite similar, we only need to define one state for both and simply give the spawned object (flame or ice particles) a tag that will define its type. Objects decide what happens to them when collision with the attacks is registered.

## Structure

## Implementation

|  |
| --- |
| public class PlayerPowerHandler : MonoBehaviour {  PlayerPower[] powers;  int currentPower = 0;  public GameObject icePowerPrefab, firePowerPrefab;  void Start() {  //set the available powers  powers = new PlayerPower[] {  new PlayerPowerTelekinesis(this.gameObject),  new PlayerPowerBreath(this.gameObject, firePowerPrefab),  new PlayerPowerBreath(this.gameObject, icePowerPrefab) };  }  // Update is called once per frame  void Update() {  if (Input.GetButtonDown("Switch Power")) {  //deactivate current power  powers[currentPower].Deactivate();  //go to next power if available, otherwise pick the first  currentPower = currentPower < powers.Length-1 ?currentPower+1 :0;  //actiate the new power  powers[currentPower].Activate();  }  powers[currentPower].Update();  }  } |

The PlayerPowerHandler class keeps track of the current state and calls its update method every frame

|  |
| --- |
| public abstract class PlayerPower {  //keep track of player object  protected GameObject player;  public PlayerPower(GameObject player) {  this.player = player;  }  //frame-by-frame logic  public abstract void Update();  //call when power is selected  public abstract void Activate();  //call when power is deselected  public abstract void Deactivate();  } |

The base class for each power contains functions that must be implemented

|  |
| --- |
| public class PlayerPowerTelekinesis : PlayerPower {  private float pullDistance = 6.0f;  GameObject pulledObject;  public PlayerPowerTelekinesis(GameObject player) : base(player) { }  public override void Activate() {  //send a message to the player to trigger sprite switches and sounds  player.SendMessage("SwitchPower", "Telekinesis");  }  public override void Deactivate() {  //make sure the pulled object doesn't return when we switch back  pulledObject = null;  }  public override void Update() {  if (Input.GetButton("Use Power")) {  GravitationalPull();  }  if (Input.GetButtonUp("Use Power")) {  GravitationalPush();  }  }  //called when power is active  private void GravitationalPull() {  //find all pullable objects in the scene  GameObject[] pullableObjects =  GameObject.FindGameObjectsWithTag("PullableObject");  //iterate through each object  for (int i = 0; i < pullableObjects.Length; i++) {  //check their distance from the player  if (Vector3.Distance(pullableObjects[i].transform.position,  player.transform.position) < pullDistance) {  pulledObject = pullableObjects[i];  //pull object towards the player  pullableObjects[i].GetComponent<Rigidbody2D>().velocity =  (player.transform.position -  pullableObjects[i].transform.position) \* 10;  }  }  }  //called when power is deactivated  private void GravitationalPush() {  if (pulledObject != null) {  //throw object away, towards the mouse cursor  pulledObject.GetComponent<Rigidbody2D>().velocity =  GlobalGuide.AimTowardsMouse(pulledObject.transform.position)  \* 10;  }  }  } |

Telekinesis pulls objects towards the player and throws them away towards the cursor

|  |
| --- |
| public class PlayerPowerBreath : PlayerPower {  private GameObject powerObject, powerPrefab;  public PlayerPowerBreath(GameObject player, GameObject icePowerPrefab)  : base(player) {  this.powerPrefab = icePowerPrefab;  }  public override void Activate() {  //trigger spreite switches, animations and sounds  player.SendMessage("SwitchPower", "Ice");  }  public override void Deactivate() {  //destroy the spawned power when we switch  GameObject.Destroy(powerObject);  }  public override void Update() {  if (Input.GetButtonDown("Use Power")) {  //spawn the power object  powerObject = (GameObject)GameObject.Instantiate(powerPrefab,  player.transform.position,  player.transform.rotation);  //make sure it follows the player position and rotation  powerObject.transform.parent = player.transform;  //place it slightly in front of the player  powerObject.transform.localPosition = new Vector2(-2.0f, 0.0f);  }  }  } |

Fire and ice attacks use the same behaviour and spawn a prefab containing animations and collision detection

|  |
| --- |
| public void OnTriggerEnter2D(Collider2D target) {  if (target.gameObject.tag == "IceAttack") {  IceDeath();  }  } |

Objects that can be killed by fire or ice check for a tag when they detect collision (built-in unity call)

# Strategy

Enemies in the game can have different behaviour depending on the situation. This means that we should be able to apply behaviour to enemies during runtime and that we should have a system that allows us to separate general behaviour from specific behaviour. Seeing as all enemies have either one of the following traits, we can easily accomplish this using the strategy pattern:

* Stationary collider object that kills the blind guy on collision
* Patrolling enemy that moves between to specified positions
* Projectile-throwing object that throws projectiles towards the blind guy
* Charing object that moves towards the blind guy at a certain speed

These behaviours can be contained within their own class, all inheriting from a base EnemyBehaviour class. Specific enemies then use these different behaviours to handle their behaviour without having to duplicate code each time two objects have the same behaviour.

## Structure

## Implementation

|  |
| --- |
| public class EnemySnowman : MonoBehaviour {  private EnemyBehaviour behaviour;  private GameObject blindGuy;  public GameObject projectile;  public Sprite moltenSprite;  private bool canSwitchAttack = false;  // Use this for initialization  void Start () {  blindGuy = GameObject.FindWithTag("Blindguy");  behaviour = new EnemyBehaviourPatrol(transform,  transform.position + transform.TransformDirection(Vector2.right)  \* 8.0f);  }    // Update is called once per frame  void Update () {  behaviour.Update();  //throw projectiles at the blind guy when he's near  if (Vector2.Distance(blindGuy.transform.position,  transform.position) < 10  && canSwitchAttack) {  behaviour = new EnemyBehaviourProjectile(transform, projectile);  canSwitchAttack = false;  }  }  void OnTriggerEnter2D(Collider2D coll) {  //switch to a collision object when we melt  if (coll.gameObject.tag == "FireAttack") {  //we can no longer attack  canSwitchAttack = false;  GetComponent<SpriteRenderer>().sprite = moltenSprite;  behaviour = new EnemyBehaviourStationaryCollider(  this.gameObject,  EnemyBehaviourStationaryCollider.CollisionType.Ice);  //make this a pullable object for telekinesis  this.gameObject.tag = "PullableObject";  GetComponent<Rigidbody2D>().isKinematic = false;  }  }  } |

The snowman only needs to know when to switch between different behaviours; the behaviour logic is handled in its own class

|  |
| --- |
| public abstract class EnemyBehaviour {  public abstract void Update();  } |

EnemyBehaviour base class only needs an Update method

|  |
| --- |
| public class EnemyBehaviourCharge : EnemyBehaviour {  private Transform transform;  private Vector2 velocity;  private float speed = 10.0f;  public EnemyBehaviourCharge(Transform callerTransform,  Vector2 chargeDestination) {  transform = callerTransform;  //calculate velocity once based on destination  velocity = (chargeDestination - (Vector2)transform.position).normalized;  }  public override void Update() {  //move with charge velocity  transform.position += (Vector3)velocity  \* Time.deltaTime  \* speed;  }  } |

Charge behaviour can be used to make an object roll towards the blind guy

|  |
| --- |
| public class EnemyBehaviourProjectile : EnemyBehaviour {  private Transform transform;  private GameObject projectile, projectilePrefab;  private Vector2 target;  private float speed = 15.0f;  private float timer;  public EnemyBehaviourProjectile(Transform callerTransform,  GameObject projectileToThrow) {  transform = callerTransform;  projectilePrefab = projectileToThrow;  Throw();  }  public override void Update() {  //throw a projectile every couple of seconds  timer -= Time.deltaTime;  if (timer <= 0) {  Throw();  }  }  private void Throw() {  timer = 5.0f;  projectile = (GameObject)GameObject.Instantiate(projectilePrefab,  transform.position,  transform.rotation);  target = GameObject.FindWithTag("Blindguy").transform.position;  }  } |

Some enemies can throw projectiles at the player. Projectile logic is handled in separate projectile scripts, because some behave differently

|  |
| --- |
| public class EnemyBehaviourStationaryCollider : EnemyBehaviour {  public enum CollisionType { Fire, Ice, Normal }  public EnemyBehaviourStationaryCollider(GameObject caller, CollisionType type) {  //apply one of three different types of deaths  switch (type) {  case CollisionType.Fire:  caller.AddComponent<SetFlameDeath>();  break;  case CollisionType.Ice:  caller.AddComponent<SetFrozenDeath>();  break;  case CollisionType.Normal:  caller.AddComponent<SetDizzyDeath>();  break;  }  }  public override void Update() {  //collision is handled by the Set<Type>Death components  }  } |

A stationary collider needs to know what kind of death it should apply on collision

|  |
| --- |
| public class EnemyBehaviourPatrol : EnemyBehaviour {  private Transform transform;  private Vector2 destination, start;  private float speed = 3.0f;  public EnemyBehaviourPatrol(Transform callerTransform,  Vector2 patrolDestination) {  transform = callerTransform;  destination = patrolDestination;  start = transform.position;  }  public override void Update () {  //move towards patrol destination  if (Vector2.Distance(transform.position, destination) > 1.0f) {  transform.position += ((Vector3)destination  - transform.position).normalized  \* Time.deltaTime  \* speed;  }  //turn around if we reach the destination  else {  Vector3 current = destination;  destination = start;  start = current;  }  }  } |

When using the patrol behaviour, the player does not need to be attacked, the enemy simply movies between two points

# Prototype

# Mediator

# Decorator

# Façade

# Observer

# Null Object