Lab8 - Gameplay, Game Mechanics, Game Balancing

COMP394-001 - Practical Game Programming
Fall 2024
Instructions

Due Date(s):

- Class Work portion: in the end of class(es)
- The challenge portion: by end-of-week's Friday, 11:59PM. (you can try to finish most of the work in class as well).

ClassWork (50%): - Follow the hands-on class work taking relevant **snapshots** in a document named **Lab8_Snapshots_{YourInitials}**.

Contents

Substitute all instances of *{YourInitials}* with the,...wait for it..., your initials :)

Challenge (50%): Continue working with the class instructions on your own til you complete the Lab8.

Gameplay, Game Mechanics and Game Balancing

Gameplay comes in many different guises, and can mean many different things. Let's implement some recipes and solutions to some of the features that we may have to build and/or use over and over again in different game engines.

1 Managing Quests

1.1 Requirements

- Create

Let's create a **quest structure**, where the player needs to complete certain **objectives** to finish a quest. We have the following requirments:

- some objectives to be **optional**, and
- some objectives to not be presented to the player until they’re achieved.
- if an objective is failed and it isn’t optional, then the entire quest is failed.

1.2 Implementation

1.2.1 Quest Example

Let's try to have a demo scene of a **quest** example to playtest our solution later:

- We’ll have a few colored blocks,
- The player has to click them to complete the quest.
- One of the blocks is forbidden
    - clicking it will fail the quest.

First, we’ll need to create the scene for our quest:

- Create four **cubes** and place them in front of the camera.
- For each of the cubes, create a **material** and add it by dragging and dropping it onto the cube in the Scene view.
- Change the **color** of the material for each cube. Make one **red**, one **green**, one **yellow**, and one **blue**. (If any of these colors are hard to tell apart, change the colors to whatever suits you best.)
- Select the main camera object, and add a **Physics Raycaster** to it.

Next, we’ll create a **canvas** that displays the status of the quest and its objectives.

Create a new canvas:

- Add a Text on the hierarchy. This adds automatically a Canvas and an EventSystem.
- Resize this object so that it can comfortably contain a paragraph of text.

1.2.2 Quest.cs

- Create a new C# script called **Quest.cs**
- To the top add the following code:, and add the following code to it:

```
#if UNITY_EDITOR
using UnityEditor;
#endif
```

We intend to also see an example of Editor tooling to create a custom Inspector view for this script, so we need the guard **UNITY_EDITOR** (as this happens only in the editor) and **using UnityEditor** (as the functionality of UnityEditor is needed only on that case). Follow the code with the definition of the Quest **ScriptableObject**:

```
// A Quest stores information about a quest: its name, and its objectives.
// CreateAssetMenu makes the Create Asset menu contain an entry that
// creates a new Quest asset.
[CreateAssetMenu(fileName = "New Quest", menuName = "Quest", order = 100)]
public class Quest : ScriptableObject
{
  // Represents the status of objectives and/or quests
  public enum Status {
    NotYetComplete, // the objective or quest has not yet been completed
    Complete,    // the objective or quest has been successfully completed
    Failed     // the objective or quest has failed
  }
  public string questName; // The name of the quest
  // The list of objectives that form this quest
  public List<Objective> objectives;
  // Objectives are the specific tasks that make up a quest.
  [System.Serializable]
  public class Objective
  {
    // The visible name that's shown the player.
    public string name = "New Objective";
    // If true, the quest can be completed without this objective
    public bool optional = false;
    // If false, the objective will not be shown to the user if it's
    // not yet complete. (It will be shown if it's Complete or Failed.)
    public bool visible = true;
    // The status of the objective when the quest begins. Usually this
    // will be "not yet complete," but you might want an objective that
```

```
    // starts as Complete, and can be Failed.
    public Status initalStatus = Status.NotYetComplete;
  }
}
```

Follow the above code with the next class definition that implements the Inspector view of the above Quest ScriptableObject structure, aptly called **QuestEditor**.

```
#if UNITY_EDITOR
// Draw a custom editor that lets you build the list of objectives.
[CustomEditor(typeof(Quest))]
public class QuestEditor : Editor {
  // Called when Unity wants to draw the Inspector for a quest.
  public override void OnInspectorGUI()
  {
    // Update the current object's (referred to as "serializedObject") pending changes
(if any).
    serializedObject.Update();
    // Draw the name of the quest (PropertyField)
    EditorGUILayout.PropertyField(serializedObject.FindProperty("questName")
            , new GUIContent("Name"));
    // Draw a header (LabelField) for the list of objectives
    EditorGUILayout.LabelField("Objectives");
    // Get the property that contains the list of objectives
    var objectiveList = serializedObject.FindProperty("objectives");
    EditorGUI.indentLevel += 1; // Indent the objectives
    // For each objective in the list, draw an entry
    for (int i = 0; i < objectiveList.arraySize; i++)
    {
      EditorGUILayout.BeginHorizontal(); // Draw a single line of controls
      // Draw the objective itself (its name, and its flags)
      EditorGUILayout.PropertyField(objectiveList.GetArrayElementAtIndex(i)
                    , includeChildren: true);
      // Draw a button that moves the item up in the list
      if (GUILayout.Button("Up", EditorStyles.miniButtonLeft, GUILayout.Width(25)))
      { objectiveList.MoveArrayElement(i, i - 1); }
      // Draw a button that moves the item down in the list
      if (GUILayout.Button("Down", EditorStyles.miniButtonMid, GUILayout.Width(40)))
      { objectiveList.MoveArrayElement(i, i + 1); }
      // Draw a button that removes (deletes) the item from the list
      if (GUILayout.Button("-", EditorStyles.miniButtonRight, GUILayout.Width(25)))
      { objectiveList.DeleteArrayElementAtIndex(i); }
      EditorGUILayout.EndHorizontal();
    }
```

```
    EditorGUI.indentLevel -= 1; // Remove the indentation
    // Draw a button at adds a new objective to the list
    if (GUILayout.Button("Add Objective")){ objectiveList.arraySize += 1; }
    serializedObject.ApplyModifiedProperties(); // Save any changes
  }
}
#endif
```

> Notice first, that the code is guarded by the same **#if UNITY_EDITOR / #endif** directive and it extends the class **Editor** from the **UnityEditor** namespace).

Notice the specific **classes**, (with methods/properties/constructor(s) in braces) coming from the **UnityEditor** namespace:

Class	Property/Method/Constructor(s)
Editor	(serializedObject, OnInspectorGUI)
CustomEditor[Attribute]	(CustomEditor)
EditorGUI	(indentLevel)
EditorGUILayout	(PropertyField, LabelField, BeginHorizontal, EndHorizontal)
GUILayout	(width, Button)
PropertyField	(PropertyField)
LabelField	(LabelField)
Button	(Button)
GUIContent	(GUIContent)
EditorStyles	(miniButtonLeft, miniButtonMid, miniButtonRight)

1.2.3 Creating a Quest Asset

Let's create a quest asset:

- Open the Assets menu and choose **Create → Quest**.
- Name the new quest **Click on the Blocks**

- Click **Add Objective**.
  - Name the new objective **Click on the red block**
  - Turn on the **Visible** checkbox.
- Repeat this process two more times—once each for the **green** and **yellow** blocks.
  - Make one of them **optional**,
  - Make one other **optional** and **not visible**.
- Add one more objective, for the **blue** block.
  - Name it **Don't click on the blue block!**
  - Set its Initial Status to **Complete**

1.2.4 QuestManager.cs

Let's set up the code that manages the quest:

- Create a new C# script called **QuestManager.cs**, and add the following code to it:

```
// Represents the player's current progress through a quest.
public class QuestStatus {
  // The underlying data object that describes the quest.
  public Quest questData;
  // The map of objective identifiers.
  public Dictionary<int, Quest.Status> objectiveStatuses;
  // The constructor. Pass a Quest to this to set it up.
  public QuestStatus(Quest questData)
  {
    // Store the quest info
    this.questData = questData;
    // Create the map of objective numbers to their status
    objectiveStatuses = new Dictionary<int, Quest.Status>();
    for (int i = 0; i < questData.objectives.Count; i += 1)
    {
      var objectiveData = questData.objectives[i];
      objectiveStatuses[i] = objectiveData.initalStatus;
    }
  }
  // Returns the state of the entire quest.
  // If all nonoptional objectives are complete, the quest is complete.
  // If any nonoptional objective is failed, the quest is failed.
  // Otherwise, the quest is not yet complete.
  public Quest.Status questStatus {
    get {
      for (int i = 0; i < questData.objectives.Count; i += 1) {
```

```
      var objectiveData = questData.objectives[i];
      // Optional objectives do not matter to the quest status
      if (objectiveData.optional) { continue; }
      var objectiveStatus = objectiveStatuses[i];
      // this is a mandatory objective
      if (objectiveStatus == Quest.Status.Failed)
      {
        // if a mandatory objective fails, the whole quest fails
        return Quest.Status.Failed;
      }
      else if (objectiveStatus != Quest.Status.Complete)
      {
        // if a mandatory objective is not yet complete,
        // the whole quest is not yet complete
        return Quest.Status.NotYetComplete;
      }
    }
    // All mandatory objectives are complete, so the quest is complete
    return Quest.Status.Complete;
  }
}
// Returns a string containing the list of objectives, their
// statuses, and the status of the quest.
public override string ToString()
{
  var stringBuilder = new System.Text.StringBuilder();
  for (int i = 0; i < questData.objectives.Count; i += 1)
  {
    // Get the objective and its status
    var objectiveData = questData.objectives[i];
    var objectiveStatus = objectiveStatuses[i];
    // Don't show hidden objectives that haven't been finished
    if (objectiveData.visible == false
      && objectiveStatus == Quest.Status.NotYetComplete)
    { continue; }
    // If this objective is optional, display "(Optional)" after its name
    if (objectiveData.optional)
    {
      stringBuilder.AppendFormat("{0} (Optional) - {1}\n"
             , objectiveData.name, objectiveStatus.ToString());
    }
    else
    {
      stringBuilder.AppendFormat("{0} - {1}\n"
```

```
                , objectiveData.name, objectiveStatus.ToString());
        }
      }
      // Add a blank line followed by the quest status
      stringBuilder.AppendLine();
      stringBuilder.AppendFormat("Status: {0}", this.questStatus.ToString());
      return stringBuilder.ToString();
    }
}
// Manages a quest.
public class QuestManager : MonoBehaviour {
  // The quest that starts when the game starts.
  [SerializeField] Quest startingQuest = null;
  // A label to show the state of the quest in.
  [SerializeField] UnityEngine.UI.Text objectiveSummary = null;
  // Tracks the state of the current quest.
  QuestStatus activeQuest;
  // Start a new quest when the game starts
  void Start () {
    if (startingQuest != null)
    { StartQuest(startingQuest); }
  }
  // Begins tracking a new quest
  public void StartQuest(Quest quest) {
    activeQuest = new QuestStatus(quest);
    UpdateObjectiveSummaryText();
    Debug.LogFormat("Started quest {0}", activeQuest.questData.name);
  }
  // Updates the quest summary label
  void UpdateObjectiveSummaryText() {
    string label;
    if (activeQuest == null) {
      label = "No active quest.";
    } else {
      label = activeQuest.ToString();
    }
    objectiveSummary.text = label;
  }
  // Called by other objects when an objective has changed status
  public void UpdateObjectiveStatus(Quest quest, int objectiveNumber
                , Quest.Status status)
  {
    if (activeQuest == null) {
      Debug.LogError("UpdateObjectiveStatus: no quest is active");
```

```
      return;
    }
    if (activeQuest.questData != quest) {
      Debug.LogWarningFormat(
        $"UpdateObjectiveStatus: quest {quest.questName} is not active. Ignoring.");
      return;
    }
    // Update the objective status
    activeQuest.objectiveStatuses[objectiveNumber] = status;
    // Update the display label
    UpdateObjectiveSummaryText();
  }
}
```

- Create a new, empty game object and add a **QuestManager** component to it.
- Drag the *quest asset* you created into its **Starting Quest** field.
- Drag the *Text* object that you set up earlier into its **Objective Summary** field.

Let's set up a class that represents a change to an objective's status and can be applied when something happens:

- Create a new C# script called **ObjectiveTrigger.cs**, and add the following code to it:

```
#if UNITY_EDITOR
using UnityEditor;
using System.Linq;
#endif
// Combines a quest, a quest's objective and Status
[System.Serializable]
public class ObjectiveTrigger
{
  // The quest that we're referring to
  public Quest quest;
  // The status we want to apply to the objective
  public Quest.Status statusToApply;
  // The location of this objective in the quest's objective list
  public int objectiveNumber;
  public void Invoke() {
    // Find the quest manager
    var manager = Object.FindObjectOfType<QuestManager>();
    // Tell it to update our objective
    manager.UpdateObjectiveStatus(quest, objectiveNumber, statusToApply);
  }
}
```

```
#if UNITY_EDITOR
// Custom property drawers override how a type of property appears in
// the Inspector.
[CustomPropertyDrawer(typeof(ObjectiveTrigger))]
public class ObjectiveTriggerDrawer : PropertyDrawer
{
  // Called when Unity needs to draw an ObjectiveTrigger property
  // in the Inspector.
  public override void OnGUI(Rect position, SerializedProperty property,GUIContent label)
  {
    // Wrap this in Begin/EndProperty to ensure that undo works
    // on the entire ObjectiveTrigger property
    EditorGUI.BeginProperty(position, label, property);
    // Get a reference to the three properties in the ObjectiveTrigger.
    var questProperty           = property.FindPropertyRelative("quest");
    var statusProperty          = property.FindPropertyRelative("statusToApply");
    var objectiveNumberProperty = property.FindPropertyRelative("objectiveNumber");
    // We want to display three controls:
    // - An Object field for dropping a Quest object into
    // - A Popup field for selecting a Quest.Status from
    // - A Popup field for selecting the specific objective from;
    //     it should show the name of the objective.
    // If no Quest has been specified, or if the Quest has no
    // objectives, the objective pop up should be empty and disabled.

    // Calculate the rectangles in which we're displaying.
    var lineSpacing = 2;
    // Calculate the rectangle for the first line
    var firstLinePos = position;
    firstLinePos.height = base.GetPropertyHeight(questProperty, label);
    // And for the second line (same as the first line, but shifted down one line)
    var secondLinePos = position;
    secondLinePos.y = firstLinePos.y + firstLinePos.height + lineSpacing;
    secondLinePos.height = base.GetPropertyHeight(statusProperty, label);
    // Repeat for the third line (same as the second line, but shifted down)
    var thirdLinePos = position;
    thirdLinePos.y = secondLinePos.y + secondLinePos.height + lineSpacing;
    thirdLinePos.height = base.GetPropertyHeight(objectiveNumberProperty, label);
    // Draw the quest and status properties, using the automatic property fields
    EditorGUI.PropertyField(firstLinePos, questProperty, new GUIContent("Quest"));
    EditorGUI.PropertyField(secondLinePos, statusProperty, new GUIContent("Status"));
    // Now we draw our custom property for the object. Draw a label on the
    //   left-hand side, and get a new rectangle to draw the pop up in
```

```
    thirdLinePos = EditorGUI.PrefixLabel(thirdLinePos, new GUIContent("Objective"));
    // Draw the UI for choosing a property
    var quest = questProperty.objectReferenceValue as Quest;
    // Only draw this if we have a quest, and it has objectives
    if (quest != null && quest.objectives.Count > 0)
    {
      // Get the name of every objective, as an array
      var objectiveNames = quest.objectives.Select(o => o.name).ToArray();
      // Get the index of the currently selected objective
      var selectedObjective = objectiveNumberProperty.intValue;
      // If we somehow are referring to an object that's not
      // present in the list, reset it to the first objective
      if (selectedObjective >= quest.objectives.Count) { selectedObjective = 0; }

      // Draw the pop up, and get back the new selection
      var newSelectedObjective = EditorGUI.Popup(thirdLinePos, selectedObjective,
                                 objectiveNames);
      // If it was different, store it in the property
      if (newSelectedObjective != selectedObjective)
      {
        objectiveNumberProperty.intValue = newSelectedObjective;
      }
    }
    else
    {
      // Draw a disabled pop up as a visual placeholder
      using (new EditorGUI.DisabledGroupScope(true))
      {
        // Show a pop up with a single entry: the string "-".
        // Ignore its return value, since it's not interactive anyway.
        EditorGUI.Popup(thirdLinePos, 0, new[] { "-" });
      }
    }
    EditorGUI.EndProperty();
  } //End OnGUI

  // Called by Unity to figure out the height of this property.
  public override float GetPropertyHeight(SerializedProperty property, GUIContent label)
  {
    // The number of lines in this property
    var lineCount = 3;
    // The number of pixels in between each line
    var lineSpacing = 2;
    // The height of each line
```

```
    var lineHeight = base.GetPropertyHeight(property, label);
    // The height of this property is the number of lines times the
    // height of each line, plus the spacing in between each line
    return (lineHeight * lineCount) + (lineSpacing * (lineCount - 1));
  }
} //End of ObjectiveTriggerDrawer
#endif
```

1.2.5 TriggerObjectiveOnClick.cs

Finally, let's set up the cubes so that they complete or fail objectives when they're clicked:

- Create a new C# script called **TriggerObjectiveOnClick.cs**, and add the following code to it:

```
using UnityEngine.EventSystems;
// Triggers an objective when an object enters it.
public class TriggerObjectiveOnClick : MonoBehaviour, IPointerClickHandler
{
  // The objective to trigger, and how to trigger it.
  [SerializeField] ObjectiveTrigger objective = new ObjectiveTrigger();
  // Called when the player clicks on this object
  void IPointerClickHandler.OnPointerClick(PointerEventData eventData)
  {
    // We just completed or failed this objective!
    objective.Invoke();
    // Disable this component so that it doesn't get run twice
    this.enabled = false;
  }
}
```

- Add a **TriggerObjectiveOnClick** component to each of the cubes.
- For each one, drag in the quest asset into its Quest field, and select the appropriate status that the objective should be set to (that is, set the blue cube to Failed, and the rest to Complete).

The canvas itself may block your raycasts unless you disable the text as a raycast target.

- Play the game. The state of the quest is shown on the screen, and changes as you click different cubes.

2 Using Machinations.io to balance games

- Create a Machinations account.
- Implement the following challenge:
  - Do the basic tutorial See video
  - Do also one other tutorial
  - Upload snapshots of the models you created both static and dynamic (while running).

3 Deliverables

- Zip and upload in eCentennial **Lab8_{YourInitials}** folder including:
  - .unitypackage of the

Summary Creating a quest system here involves:

- creating something to be a quest (the cubes),
- a UI to show the status of the quest, and
- some actual quests.

Intro to Machinations.io:

- account
- tutorials
- using it to balance a Game

> When you're building a system like this, it's important to think through the different combinations of states that the objects can be involved in. Think about what a mischievous, malicious, confused, or unlucky player might do: they might do things out of order, skip over content, or find ways to do what your code doesn't expect.

formatted by Markdeep 1.17