**Rafał Nazarko, Jan Maślanka**

Koło Naukowe Elektroniki i Technologii Informacyjnych

**Symulator konstruowania i testowania obwodów elektronicznych w wirtualnej rzeczywistości**

**Streszczenie**

Artykuł opisuje proces projektowania, działania i etapy rozwijania projektu realizowanego przez Koło Naukowe Elektroniki i Technologii Informacyjnych, opierającym się na tworzeniu układów elektronicznych w świecie wirtualnej rzeczywistości.

**Słowa kluczowe:** wirtualna rzeczywistość, symulator, układ elektroniczny.

1. Wprowadzenie

Głównym przeznaczeniem symulatora jest możliwość tworzenia i testowania obwodów elektronicznych, poznania ich działania, procesów w nich panujących oraz opanowania podstawowej wiedzy niezbędnej do zrozumienia świata elektroniki.

Z racji na przystępny i ciekawy sposób przekazania wartości edukacyjnych, symulator ten idealnie nadaje się do prezentacji na imprezach pokazowych dla różnorodnych kategorii wiekowych. W artykule zostanie dogłębnie omówiony proces projektowy oraz zasady panujące w jego wirtualnym środowisku.

1. Środowisko uruchomieniowe symulatora

Podstawą działania symulatora jest popularne, zintegrowane środowisko do tworzenia gier komputerowych, zarówno trójwymiarowych jak i dwuwymiarowych, o nazwie Unity. Wykorzystanie tego silnika do naszych potrzeb jest umożliwione na podstawie darmowej licencji.

Językiem programowania, który obsługuje użytkowany przez nas silnik, jest potomek gatunku rodziny C, dokładniej – C# (ang. *C Sharp*, czyt. *„si szarp”*). Jest zorientowany w sposób obiektowy co w dużej mierze ułatwia posługiwanie się Unity.

1. Tworzenie obiektów trójwymiarowych

Do stworzenia obiektów trójwymiarowych, wykorzystany został program Fusion360 udostępniony na licencji studenckiej. Oferuje on bardzo przejrzyste i intuicyjne kreowanie obiektów, na podstawie narysowanych szkiców.

W celu jak największej immersyjności świata wirtualnego, wszystkie obiekty zostały przeniesione ze świata rzeczywistego z zachowaniem ich realnych wymiarów i kształtów. Z posiadanych elementów układów elektronicznych takich jak diody, silniki, baterie, przyciski pobrane zostały wymiary za pomocą dokładnych narzędzi pomiarowych. Na ich podstawie zostały zbudowane pierwotne szkice, które w dalszym etapie wykorzystano do stworzenia w właściwych modeli wykorzystywanych w symulatorze. Ruchome elementy zostały wyodrębnione dla zwiększenia funkcjonalności podczas wykorzystywania ich w silniku.

Z racji na duże zużycie mocy obliczeniowej podczas korzystania z symulatora, siatki wszystkich obiektów zostały możliwie jak najbardziej uproszczone, pozwalając znacząco poprawić wydajność i płynność rozgrywki oraz zwiększyć komfort przebywania w rzeczywistości wirtualnej.

Następnie do każdego wielokąta siatki przypisano odpowiedni materiał określający kolor, chropowatość powierzchni, zdolność do pochłaniania światła oraz metaliczność. Ostatnim elementem było wyeksportowanie gotowych elementów wraz z ich materiałami do plików z rozszerzeniem *.fbx*, które z łatwością adaptują się do różnorodnych środowisk graficznych, a w dalszej kolejności zaimportowanie ich do projektu symulatora w silniku Unity.

1. Prefabrykaty

Prefabrykaty są to odpowiednio zdefiniowane obiekty w silniku graficznym, które są często wykorzystywane podczas rozgrywki. Zawierają szereg komponentów zawierających właściwości oraz skrypty. Można powiedzieć, że jest to martyca, na podstawie której Unity tworzy ich kopie i umieszcza w wirtualnym świecie symulatora. Odpowiednie zdefiniowanie prefabrykatów pozwala znacząco uprościć proces twórczy, zwiększyć wydajność oraz czytelność kodu.

Na potrzebę symulatora zostały stworzone prefabrykaty dla wszystkich elementów obwodu elektrycznego. Każdy z nich posiada klasę *ItemClass*. Informacje w nich zawarte określają wielkość, ilość zajmowanego miejsca, nazwę, model trójwymiarowy itp.

|  |
| --- |
| public class ItemClass : MonoBehaviour  {  public Sprite itemImage;  public string itemName;  public Vector3Int size;  public Vector3 originTransform = Vector3.down;  } |

Listing 1. Definicja klasy ItemClass

Tabela 1. Przykładowy prefabrykat baterii 9V

|  |  |  |
| --- | --- | --- |
| Komponent | Nazwa parametru | Wartość parametru |
| Transform | Position | 0,0,0 (Vector3) |
| Rotation | 0,0,0 (Vector3) |
| Scale | 1,1,1 (Vector3) |
| Item Class (Script) | Item Image | Battery 9V (Splash) |
| Item Name | Battery 9V (String) |
| Size | 2,1,0 (Vector3) |
| Origin Transform | 0,-1,0 (Vector3) |
| Rigidbody | Mass | 0,1 (Float) |
| Drag | 0,5 (Float) |
| Angular Drag | 0,5 (Float) |
| Use Gravity | false (Boolean) |
| Is Kinematic | false (Boolean) |

Źródło: opracowanie własne.

Tak zdefiniowane obiekty są gotowe do przeniesienia bezpośrednio na obszar, w który znajduje się użytkownik lub mogą być także zaczerpnięte do skryptów innych obiektów, np. do menu wyboru przedmiotu tzw. ekwipunku.

1. Interfejs

Z uwagi na to, że użytkownik przez cały okres pobytu w symulatorze musi mieć na głowie założone okulary do wirtualnej rzeczywistości, interfejs musi być ograniczony do minimum. Zabieg ten pozwala zwiększyć komfort użytkowania oraz pozwala skupić się na samej rozgrywce.

W myśl tego, interfejs w tym symulatorze jest widoczny jedynie podczas przeglądania ekwipunku. Minimalistyczny styl jasno wskazuje na możliwe pola wyboru. Podczas gdy jest widoczny, czas w symulatorze ulega zatrzymaniu. Wszystkie widoczne opisy są w języku angielskim, aby dostosować go każdego odbiorcy i ułatwić korzystanie z niego. Prosta, przewijana lista zawiera obrazy elementów obwodu (zaczerpniętych z ich prefabrykatów), w równomiernym rozmieszczeniu pięć w wierszu. Lista ta jest tworzona w oparciu o komponent ItemList, zawartego w jednej z części składowych interfejsu. Jest postać wygląda następująco:

Tabela 2. Postać listy przedmiotów

|  |  |
| --- | --- |
| Indeks listy | Przedmiot (GameObject) |
| 0 | Battery 9V |
| 1 | Button |
| 2 | Capacitor |
| 3 | DC Motor |
| 4 | Diode |
| 5 | LiPo 7.4V |
| 6 | Mini Servo |
| 7 | Resistor |
| 8 | Switch |
| 9 | Transistor |

Źródło: opracowanie własne.

Po wybraniu interesującego użytkownika elementu, ekwipunek jest zamykany, czas ponownie zaczyna upływać a przed użytkownikiem pojawia się wybrany element obwodu.

|  |
| --- |
| public void OnItemSelected(int index) //Fires when user select an object to place  {  if(selection.transform.childCount > 2)  {  Destroy(selection.transform.GetChild(2).gameObject); //Destroy previous selection object  }  var selectionController = selection.GetComponent<SelectionController>();  selectionController.ToggleState(true); //Show placeholder  var playerController = player.GetComponent<PlayerController>();  playerController.OpenInventory(); //Hide inventory  GameObject selectedItem = Instantiate(itemList[index]); //Create new object in scene  selectionController.SetPlaceholderSize(selectedItem.GetComponent<ItemClass>().size);  selectedItem.transform.position = selection.transform.position; //Set the same postion as parent  selectedItem.transform.rotation = selection.transform.rotation; //Set the same rotation as parent  selectedItem.transform.SetParent(selection.transform); // Add this object to selection plane  playerController.ChangePlacing(true); //Change state to placing item  } |

Listing 2. Fragment kodu wywoływany po dokonaniu wyboru elementu

1. Obszar roboczy

Obszar roboczy jest to wydzielona część sceny, w której znajduje się użytkownik, po której można się swobodnie poruszać oraz budować na niej obwód elektroniczny. Jest on zbudowany z płaskiej płyty, która z kolei jest złożona z kwadratowych modułów reprezentujących miejsce o jednostkowej długości.

Jak wcześniej zostało wspomniane, po wyborze interesującego elementu z ekwipunku, pojawia się on w tej przestrzeni. Użytkownik może ustawić go w dowolnym, niezajętym miejscu. O tym, czy takie rozłożenie jest możliwe informuje kolor półprzezroczystego prostopadłościanu otaczającego wybrany obiekt zwanego *Placeholder’em*. Jest on wielkości odpowiadającej ilości potrzebnego wolnego miejsca, różnej w zależności od danego obiektu, która oczywiście jest zdefiniowana w klasie *ItemClass*. Jeśli w obrębie *Placeholder’a* nie znajdują się żadne wcześniej umieszczone elementy układu, to przyjmuje on kolor zielony co oznacza, że w tym miejscu można umieścić obiekt ostatnio wybrany z ekwipunku. W przeciwnym wypadku kolor jest zmieniany na czerwony co świadczy o braku takiej możliwości. Proces ten realizuje skrypt *ColliderState* przypisany jako komponent do *Placeholder’a*. Jeśli dowolny obiekt o tagu *„Placed”* wejdzie w obszar zajmowany przez *Placeholder’a*, to zmienna publiczna *isSomethingWithin* przyjmuje wartość *prawda* a obiekt zostaje przypisany do zmiennej publicznej *objectIn* typu *GameObject*. Jeśli obiekt ten opuści wcześniej zajmowany obszar to obydwie zmienne powracają do stanu podstawowego. W ten sposób możliwe jest stałe kontrolowanie elementów znajdujących się w obszarze *Placeholder’a*, niezależnie od wielkości jaką przyjmuje.

|  |
| --- |
| public class ColliderState : MonoBehaviour  {  public bool isSomethingWithin;  public bool isSelectionPlaneWithin;  public GameObject objectIn;  private void OnTriggerStay(Collider other) //Fires when rigitbody get into collider  {  if(other.CompareTag("Placed")) //Only items with tag "Placed" can affect  {  objectIn = other.transform.parent.gameObject;  isSomethingWithin = true;  }  else if(other.CompareTag("SelectionPlane")) //If selection plane is in collider  {  isSelectionPlaneWithin = true;  }  }  private void OnTriggerExit(Collider other) //Fires when rigitbody exit collider  {  objectIn = null;  if (other.CompareTag("Placed")) //Only items with tag "Placed" can affect  {  isSomethingWithin = false;  }  else if (other.CompareTag("SelectionPlane")) //If selection plane is in collider  {  isSelectionPlaneWithin = false;  }  }  } |

Listing 3. Skrypt ColliderState

Podczas przebywania w trybie ustawiania, możliwe jest obracanie obiektu w jednym z czterech kierunków. Potwierdzając operację, obiekt zostaje ustawiony w wyznaczonym miejscu, a użytkownik wraca do trybu podstawowego, w którym znajdował się przed wywołaniem menu wyboru elementów.

Elementy obwodu mogą być umieszczane tylko i wyłącznie w taki sposób, aby zajmowane przez nie miejsca, idealnie pokrywały się z modułami obszaru roboczego. W istocie oznacza to, że ustawiany obiekt jest przyciągany przez najbliższy moduł i nie da się go postawić w pełni swobodnie. Poniższy schemat pozwoli zrozumieć zasadę ustawiania elementów:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

Rysunek 1. Schemat poprawnego rozmieszczenia elementów układu w obszarze roboczym (linią przerywaną oznaczone są moduły obszaru roboczego, ciągłą - ustawione obiekty)

Źródło: opracowanie własne

Usuwanie umieszczonych obiektów przebiega w bardzo podobny sposób jak proces ustawiania. Należy aktywować tryb usuwania, po czy skierować wskaźnik na element układu, który chcemy usunąć. Podczas przebywania w tym trybie, *Placeholder* będzie miał kolor czerwony. Jego rozmiar zmienia się w zależności od tego, na jaki element aktualnie wskazuje użytkownik. Po potwierdzeniu wybranej operacji, zaznaczony obiekt zostaje usunięty, jednak w przeciwieństwie do trybu ustawiania, nie wraca się do trybu podstawowego.

1. Łączenie elementów w obwód
2. Pomiar wielkości fizycznych

Literatura

1. Lis M., *C#. Praktyczny kurs. Wydanie III*, Wydawnictwo Helion, 2016

Źródła internetowe