**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 matryca, 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. Jej 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 zwany *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

W implementacji łączenia elementów należy uwzględnić metodę, która będzie użyta do policzenia wartości natężenia i potencjału na każdym z elementów układu. W przypadku układów prądu stałego dobrze sprawdzi się metoda potencjałów węzłowych, więc potrzebna jest klasa do przedstawienia węzłów oraz lista tej klasy.

|  |
| --- |
| public class CircuitController : MonoBehaviour  {  public static CircuitController instance;  public Material connectionMaterial;  void Awake()  {  instance = this;  }  private class Node  {  public List<ConnectionClass> connections = new List<ConnectionClass>();  public List<LineRenderer> lines = new List<LineRenderer>();  public int id;  public Node(ConnectionClass conn)  {  connections.Add(conn);  conn.connected = true;  }  }  } |

Listing 4. Klasa CircuitController

Każdy węzeł posiada listę połączeń, które łączy. Połączenia przedstawione są w klasie *ConnectionClass*.

|  |
| --- |
| public class ConnectionClass : MonoBehaviour  {  public enum ConnectionType  {  Plus,  Minus,  Pin1,  Pin2,  Pin3,  Pin4  }  public ItemClass element;  public ConnectionType type;  public bool connected = false;  } |

Listing 5. Klasa ConnectionClass

Nowe połączenia powstają gdy użytkownik włączy tryb łączenia, każde nowe włączenie tego trybu dodaje nowy węzeł. Użytkownik wybiera komponenty, które chce połączyć, z wybranych komponentów pobierane jest połączenie najbliższe kursorowi. Wybieranie elementów do połączenia w znacznej części podobne jest jak przy ich usuwaniu, element który znajduje się w obiekcie *Placeholder* zostaje przekazany do funkcji dodającej ją do listy węzłów.

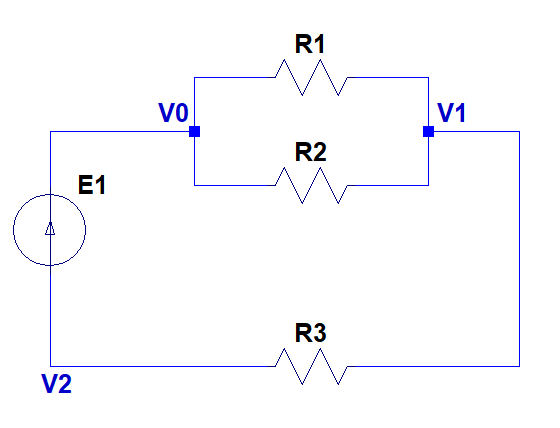
Pomiędzy połączeniami tworzone są linie reprezentujące przewody. W przypadku gdy węzeł łączy więcej niż dwa połączenia linie (przewody), poprowadzone są od środka linii łączącej pierwsze dwa połączenia.

1. Pomiar wielkości fizycznych

Po powstaniu nowego połączenia układ sprawdzany jest pod względem, rozwiązywalności. Układ rozwiązywalny to taki, w którym każde połączenie jest przypisane do jakiegoś węzła, jeśli układ spełnia ten warunek to z połączeń zawartych w liście węzłów wybierane są komponenty należące do układu. Do zmodyfikowanej metody węzłowej potrzebujemy wypełnić macierze, z których obliczone zostaną potencjały na węzłach.

* 1. Metoda potencjałów węzłowych

Metoda pozwalająca na znalezienie napięć między węzłami. Tworzone są równania na bazie Praw Kirchhoffa dla każdego węzła oprócz węzła zerowego, który zostaje uziemiony.



Rysunek 2. Schemat przykładowego obwodu

Źródło: opracowanie własne

Na przykładzie:

Natężenie na rezystorze R1:

Natężenie na rezystorze R2:

Natężenie na rezystorze R3:

Z prawa Kirchhoffa:

* 1. Zmodyfikowana metoda potencjałów węzłowych

Rozwinięcie metody potencjałów węzłowych pozwalające na nie tylko obliczenie napięć na węzłach ale także niektórych natężeń gałęziowych (wychodzących ze źródeł napięć).

* 1. Zmodyfikowana metoda potencjałów węzłowych - tworzenie macierzy

Zmodyfikowana metoda potencjałów węzłowych sprowadza się do rozwiązania równania macierzowego:

gdzie:

* Macierz *A* składająca się z czterech macierzy,

gdzie:

* + *G* – macierz konduktancji połączonych rezystorów,
  + *B* – macierz połączeń z źródłami napięciowymi,
  + *C* – podobna do macierzy *B* ale uwzględnia źródła prądowe,
  + *D* – macierz zer.
* Macierz *x* – macierz w której znajdą się wyniki,
* Macierz *b*– macierz stworzona z dwóch macierzy:
  + *i* – Macierzy z natężeniami ze źródeł prądowych na każdej z gałęzi,
  + *v* – Macierzy z napięciami ze źródeł napięć.

Równanie dla wcześniej podanego przykładu to:

gdzie:

* *V1* – potencjał na pierwszym węźle,
* *V2* – potencjał na drugim węźle,
* *IE1* – natężenie prądu pobierane ze źródła napięciowego,
* *E1* – napięcie na źródle napięcia.

Aby wypełnić macierze najpierw obliczane są ich rozmiary, macierz *A* jest macierzą kwadratową stopnia równego liczbie węzłów (nie licząc węzła zerowego, który uziemiamy), macierze *b* i *z* są macierzami prostokątnymi o liczbie wierszy równej liczbie węzłów pomnożonych przez liczbę źródeł napięcia. Dla każdego z elementów listy *components* sprawdzany jest jego typ (rezystor, źródło), pobrana wartość (rezystancje, napięcie) i wstawiona w odpowiednie miejsce w macierzy. *A[high-1, high-1], A[low-1, low-1]* modyfikują przekątną macierzy *G, A[high - 1, low - 1], A[low - 1, high - 1]* modyfikują elementy poza przekątną w macierzy *G*. *A[high - 1, index], A[index, high - 1], A[high - 1, index], A[index, high - 1]* modyfikują macierze *B* i *C*. Macierz *D* jest macierzą kwadratową stopnia równego liczbie źródeł napięcia.

|  |
| --- |
| public (Matrix<float> A, Vector<float> b) CalculateMatrices(List<ItemClass> components, int sources)  {  var matrixSize = nodes.Count + sources - 1;  var A = Matrix<float>.Build.Dense(matrixSize, matrixSize);  var b = Vector<float>.Build.Dense(matrixSize);  var index = matrixSize - sources;  foreach (var component in components)  {  var high = component.high;  var low = component.low;  if (component.type == ItemClass.ItemType.Resistor)  {  var value = component.GetComponent<ResistorClass>().resistance;  if (high != 0)  {  A[high - 1, high - 1] += 1/value;  }  if (low != 0)  {  A[low - 1, low - 1] += 1/value;  }  if (high != 0 && low != 0)  {  A[high - 1, low - 1] -= 1/value;  A[low - 1, high - 1] -= 1/value;  }  }  else if (component.type == ItemClass.ItemType.Source)  {  var value = component.GetComponent<SourceClass>().voltage;  if (high != 0)  {  A[high - 1, index] += 1;  A[index, high - 1] += 1;  }  if (low != 0)  {  A[low - 1, index] -= 1;  A[index, low - 1] -= 1;  }  b[index] = value;  index = index + 1;  }  }  return (A, b);  } |

Listing 6. Metoda CalculateMatrices klasy CircuitController

* 1. Zmodyfikowana metoda potencjałów węzłowych – rozwiązywanie równania

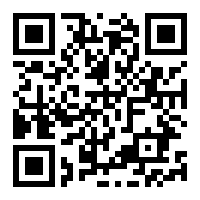
W celu rozwiązania równania macierzowego została użyta funkcja z biblioteki MathNet.Numerics. Jest ona metodą macierzy *A*. Z obliczonych potencjałów i natężeń można obliczyć natężenia i spadki napięć na reszcie elementów i wyświetlić je użytkownikowi.

|  |
| --- |
| var x = A.Solve(b); |

Listing 7. Rozwiązanie równania macierzowego

9. Podsumowanie

Aktualnie symulator ma wąski zakres zastosowań, jednak w przyszłości planujemy możliwość symulowania układów prądu zmiennego co wymagałoby dodania rozwiązywania równania przez metodę Newtona-Raphsona. Innym planem na rozwinięcie projektu jest dodanie elementów z których można byłoby tworzyć pojazdy co z pewnością spodobałoby się odwiedzającym stoisko naszego koła podczas różnych pokazów. Kod źródłowy opisywanego symulatora można znaleźć pod adresem: https://github.com/jaenek/VR-Elektronika/.



Rysunek 3. Kod QR zawierający adres do repozytorium projektu

Źródło: opracowanie własne

Literatura

1. Lis M., *C#. Praktyczny kurs. Wydanie III*, Wydawnictwo Helion, 2016
2. Papoulis A., *Obwody i układy*, WK Ł, Warszawa., 1988
3. Zieliński T., *Od teorii do cyfrowego przetwarzania sygnałów*., Wydział EAIiE AGH, Kraków., 2002

Źródła internetowe

1. Stefan Jahn, Modified Nodal Analysis, http://qucs.sourceforge.net/tech/node14.html

2. https://numerics.mathdotnet.com/LinearEquations.html

3. https://en.wikipedia.org/wiki/Nodal\_analysis

4. https://en.wikipedia.org/wiki/Modified\_nodal\_analysis