LZWPlib v2.9

&

Laboratorium Zanurzonej Wizualizacji Przestrzennej

mgr inż. Robert Trzosowski
robtrzos@pg.edu.pl

SPIS TREŚCI

1. LZWPlib

Pakiet LZWPlib jest paczką typu unitypackage, której celem jest ułatwienie przystosowania aplikacji tworzonej z użyciem silnika gier Unity do działania w środowisku typu CAVE poprzez podstawową obsługę elementów specyficznych dla tego rodzaju systemów rzeczywistości wirtualnej.

Pakiet przeznaczony jest dla środowiska Unity w wersji 5.1 – 2018.1.9.

1.1. *Zawartość pakietu*

Głównym elementem pakietu LZWPlib są dwie wtyczki – biblioteki DLL: pierwsza zawiera funkcje odpowiadające za prawidłową pracę aplikacji w środowisku CAVE, druga zaś jest rozszerzeniem dla samego edytora, dodaje do niego m.in. nowe okno z ustawieniami. Poza tym pakiet ten zawiera: prefabrykaty pozwalające szybko dodać podstawową integrację z systemem typu CAVE, edytowalne skrypty (np. obsługujące przemieszczanie się) i programy cieniujące, obiekty przechowujące konfigurację poszczególnych stanowisk dostępnych w LZWP, szablon podstawowej sceny z umieszczonymi na niej wymaganymi elementami, przykładowe sceny demonstrujące użycie poszczególnych funkcji pakietu. Zarówno biblioteki jak i edytowalne skrypty napisane zostały w języku C#, natomiast programy cieniujące – w języku ShaderLab.

Po zaimportowaniu pakietu LZWPlib do projektu Unity w katalogu zasobów pojawia się następująca zawartość (wymieniono jedynie ważniejsze pozycje):

- *LZWPlib* – główny katalog pakietu
  - *Audio* – pliki audio używane przez skrypt przemieszczania się
  - *Editor* – pliki dostępne jedynie dla edytora, nie będą dołączane do wynikowej aplikacji
    - *Plugins* – wtyczka przeznaczona dla edytora
    - *Resources* – zasoby: ikony, obiekty ustawień, obiekt metadanych aplikacji
      - *Config* – obiekty typu *ScriptableObject* przechowujące różne wersje podstawowej konfiguracji, które można wykorzystać podczas testów w edytorze
      - *CustomConfig* – obiekty typu *ScriptableObject* przechowujące różne wersje konfiguracji dodatkowej (specyficznej dla konkretnej aplikacji), które można wykorzystać podczas testów w edytorze
    - *Scripts* – edytowalne skrypty edytora
  - *Examples* – sceny demonstrujące implementację różnych funkcji pakietu
    - *Custom* config – opcje konfiguracyjne aplikacji
    - Fingertracking - użycie systemu śledzenia dłoni i palców
    - *Input* – obsługa wejścia (kontrolery Flystick)
    - *Logging* – pisanie do logu / konsoli
    - *Mocap* – użycie systemu śledzenia ciała
    - *Switch scene* – przełączanie scen
    - *Sync* – synchronizacja elementów świata
    - *Sync time* – synchronizacja czasu

        - *Tracking* – obsługa systemu śledzenia
        - *_SharedAssets* – zasoby współdzielone pomiędzy przykładowymi scenami
    - *ObstacleWallWarning* – pliki potrzebne do wyświetlania ostrzeżeń o zbliżaniu się do przeszkody fizycznej
    - *PhysicsMaterials* – materiały fizyczne (dla obiektów kolizji)
    - *Plugins* – wtyczka przeznaczona dla docelowej aplikacji oraz dla edytora
    - *Prefabs* – podstawowe prefabrykaty
        - *[LZWPlib].prefab* – obiekt główny, odpowiadający m.in. za inicjalizację
        - *[LzwpOrigin].prefab* – prefabrykat reprezentujący umiejscowienie jaskini w świecie wirtualnym
        - *[LzwpOrigin_M].prefab* – jak wyżej, dodatkowo dający możliwość przemieszczania się po świecie wirtualnym – chodząc lub latając
        - *[CameraContainer].prefab* – kontener z dwoma kamerami
        - *TrackedObject.prefab* – obiekt wirtualny, który ma podążać za rzeczywistym, śledzonym obiektem
        - *[EditConfigInPlayMode].prefab* – obiekt pozwalający edytować konfigurację w trybie gry (ang. *Play mode*)
    - *SceneTemplate* – szablon podstawowej sceny
    - *Scripts* – edytowalne skrypty
    - *Shaders* – programy cieniujące lub ich fragmenty
- *UnityPackages*
    - *JsonNet* – port biblioteki Json.NET dla Unity (używany do serializacji i deserializacji obiektów konfiguracyjnych)
    - *JsonNetSample*

1.2. *Założenia ogólne*

Zakładamy, że aplikacja będzie zazwyczaj działała na klastrze komputerowym w wielu instancjach, wśród których jedna będzie pełniła rolę autorytarnego serwera, który będzie obsługiwał dane wejściowe (w tym dane z systemu śledzenia), na którym przeprowadzane będą obliczenia (np. fizyczne) i wykonywana będzie cała logika aplikacji, oraz który aktualizować będzie stan aplikacji i synchronizować go z pozostałymi instancjami (synchronizowane muszą być te rzeczy, które wpływają w jakiś sposób na rezultat wizualny działania aplikacji). Pozostałe instancję będą z kolei pełniły rolę klientów, odpowiedzialnych w głównej mierze za odpowiednie wygenerowanie obrazu do wyświetlenia na ścianach jaskini. Dopuszczamy jednak możliwość uruchomienia kilku instancji aplikacji na pojedynczym komputerze w osobnych oknach – będzie to przydatne podczas testowania synchronizacji stanu. Możliwe też będzie wykorzystanie tylko jednej instancji aplikacji, która będzie działała tak jak serwer, a przy tym renderowała obraz dla jednego lub kilku ekranów.

Przyjmujemy również założenie, że 1 jednostka na wirtualnej scenie odpowiada 1 metrowi w rzeczywistości (ogólnie przyjęty standard w środowisku Unity).

1.3. *Okno edytora*

Po pierwszym zaimportowaniu pakietu LZWPlib w danym edytorze Unity automatycznie otwarte zostaje nowe okienko narzędziowe, opatrzone nazwą tożsamą z nazwą pakietu. W dowolnej chwili otworzyć je można samemu, korzystając z pozycji *Window – LZWPlib* w menu głównym edytora, bądź też z nowo dodanej do tego menu pozycji *LZWPlib – Editor Window*.

Okno to zawiera cztery zakładki:

- *Project settings* – zawiera listę wymaganych i opcjonalnych ustawień projektu związanych z jego działaniem w środowisku CAVE, a także pola do podpięcia obiektów konfiguracji – głównej i dodatkowej;
- *App metadata* – umożliwia wprowadzenie danych opisujących tworzoną aplikację;
- *Input* – podczas działania aplikacji w edytorze wyświetla dane odebrane od systemu śledzenia;
- *Testing* – zawiera opcje pomagające w testowaniu synchronizacji wielu instancji aplikacji.

1.4. *Ustawienia projektu*

Do poprawnego działania aplikacji w jaskiniach rzeczywistości wirtualnej konieczna jest modyfikacja domyślnych ustawień projektu. Ustawienia wymagane i opcjonalne przedstawia lista dostępna w zakładce *Project settings* okna *LZWPlib* (rys. 1.1). Pozycje oznaczone zielonym znakiem *V* są ustawione poprawnie, oznaczone czerwonym znakiem *X* są ustawione niepoprawnie i wymagają zmiany, zaś oznaczone żółtym znakiem *X* są opcjonalne i zalecana jest ich zmiana. Poszczególne ustawienia można skorygować używając odpowiadającego im przycisku *Fix* lub *Toggle* po prawej stronie okna (niektóre z nich – te z przyciskiem *Toggle* – można przełączyć z powrotem). Jeśli lista zawiera nieprawidłowo ustawione pozycje wymagane, u jej spodu znajdować się będzie przycisk *Adjust all settings*, którego kliknięcie spowoduje skorygowanie wszystkich tych ustawień. Twórca aplikacji może chcieć skorygować wybrane ustawienia później (być może z poziomu kodu), dlatego LZWPlib nie robi tego automatycznie po zaimportowaniu pakietu.

Lista ustawień zawiera następujące pozycje (odpowiadające opcjom w oknie *PlayerSettings*):

- *Virtual Reality Supported* – włączenie obsługi rzeczywistości wirtualnej jest wymagane; odpowiada opcji *XR Settings – Virtual Reality Supported*;
- *Enabled VR SDKs* – tryb stereo musi znajdować się na początku listy obsługiwanych przez aplikację trybów; odpowiada opcji *Stereo Display (non head-mounted)* na liście *XR Settings – Virtual Reality SDKs*;
- *Graphics API for Windows* – lista API graficznych obsługiwanych przez aplikację musi zawierać: Direct3D 11, OpenGL Core, Direct3D 12, Vulkan; odpowiada to liście *Other Settings – Rendering – Graphics APIs for Windows* przy odznaczonym polu *Auto Graphics API for Windows*;
- *Run in background* – wymagana jest możliwość działania aplikacji w tle; odpowiada opcji *Resolution and Presentation – Resolution – Run In Background*;

- *Visible in background* – wymagana jest możliwość wyświetlania aplikacji w tle przy włączonym okienkowym trybie pełnoekranowym; odpowiada opcji *Resolution and Presentation – Standalone Player Options – Visible In Background*;
- *Display Resolution Dialog* – wymagane jest domyślne ukrycie okna wyboru rozdzielczości przy starcie aplikacji; odpowiada opcji *Hidden By Default* na liście *Resolution and Presentation – Standalone Player Options – Display Resolution Dialog*;
- *Show SIpash Screen* – zalecane jest wyłączenie ekranu powitalnego (domyślnie zawierającego logo Unity); odpowiada opcji *Splash Image – Splash Screen – Show Splash Image*; jest to ustawienie opcjonalne, gdyż wymaga posiadania licencji Unity Pro.

Zakładka *Project settings* okna *LZWPlib* zawiera również dwa pola, w których należy ustawić odpowiednie obiekty konfiguracyjne, które zostaną użyte podczas budowania aplikacji:

- *Startup config* – obiekt ustawień głównych, które tymczasowo załączane są przy budowaniu aplikacji;
- *Custom config* – obiekt ustawień dodatkowych danej aplikacji z ustawionymi wartościami domyślnymi.

Rys. 1.1. *Project settings* - zakładka ustawień projektu w oknie *LZWPlib*

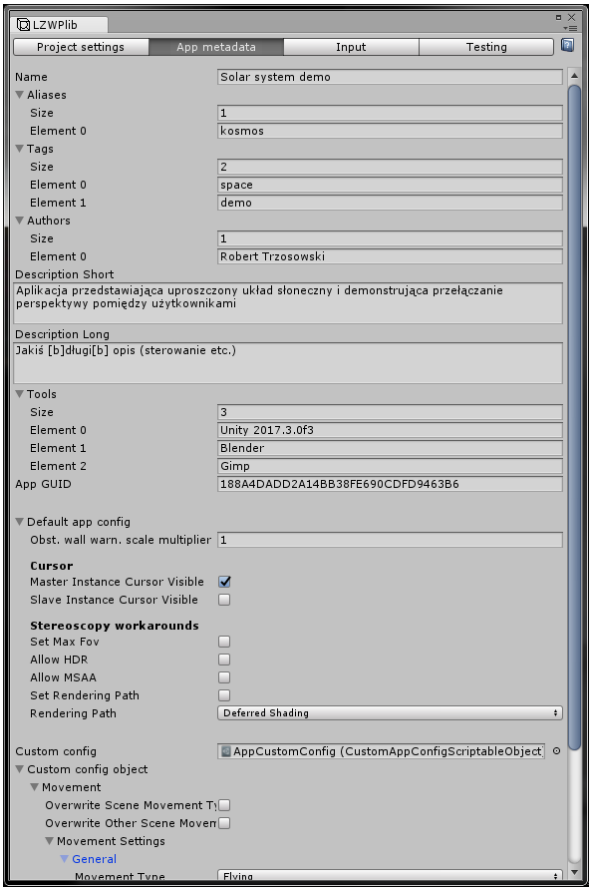
1.5. *Metadane aplikacji*

Zakładka *App metadata* okna *LZWPlib* (rys. 1.2) zawiera pola umożliwiające opisanie aplikacji oraz podanie jej domyślnej konfiguracji – podane wartości będą wykorzystywane przez moduł aplikacji usługi LzwpManager. Uzupełnienie brakujących wartości nie jest wymagane.

Wspomniane pola to:

- *Name* – nazwa aplikacji; domyślnie ustawiona na nazwę produktu (pole *Product Name* w oknie *PlayerSettings*), która z kolei domyślnie ustawiana jest na nazwę projektu przy jego tworzeniu;
- *Aliases* – alternatywne nazwy aplikacji – można tu podać np. nazwy uproszczone lub potoczne, będą one uwzględniane przy wyszukiwaniu aplikacji w panelu usługi LzwpManager;
- *Tags* – lista słów kluczowych opisujących aplikację;
- *Authors* – lista autorów aplikacji;
- *Short Description* – krótki opis aplikacji;
- *Long Description* – długi opis aplikacji;
- *Tools* – lista ważniejszych narzędzi wykorzystanych przy tworzeniu aplikacji; domyślnie do listy tej dodawany jest edytor Unity;
- *App GUID* – unikatowy identyfikator aplikacji; domyślnie przyjmuje wartość wygenerowaną podczas importu pakietu;
- *Default app config* – domyślne wartości parametrów konfiguracyjnych:
  - *Obstacle walls warning scale multiplier* – mnożnik skali ostrzeżenia o zbliżaniu się do przeszkody fizycznej;
  - *Cursor* - wyświetlanie kursora myszy:
    - *Master Instance Cursor Visible* – wyświetlanie kursora w oknie instancji głównej;
    - *Slave Instance Cursor Visible* – wyświetlanie kursora w oknach pozostałych instancji;
  - *Stereoscopy workarounds* – obejścia problemów ze stereoskopią:
    - *Set Max Fov* – ustawienie maksymalnego pola widzenia (zapobiega wycinaniu obiektów);
    - *Allow HDR* – odznaczenie tej opcji powoduje wymuszenie wyłączenia HDR (ang. *High Dynamic Range*) we wszystkich kamerach;
    - *Allow MSAA* – odznaczenie tej opcji powoduje wymuszenie wyłączenia MSAA (ang. *multisample anti-aliasing*) we wszystkich kamerach;
    - *Set Rendering Path* – wymuszenie ustawienia ścieżki renderowania we wszystkich kamerach na wartość określoną w kolejnym polu;
    - *Rendering Path* – wymuszana ścieżka renderowania (jeśli wymuszanie zostało włączone w poprzednim polu);
- Custom config – obiekt ustawień dodatkowych danej aplikacji z ustawionymi wartościami domyślnymi (dodatkowo poniżej tego pola możliwa jest edycja tych wartości).

Dodatkowo podczas budowania aplikacji do jej metadanych dopisany zostaje parametr *buildGUID*, będący unikatowym identyfikatorem danej wersji aplikacji. Metadane serializowane są do pliku *LzwpApp.json*.



Rys. 1.2. *App metadata* - zakładka metadanych aplikacji w oknie *LZWPlib*

1.6. *Moduł główny*

Każda scena zawierać powinna obiekt z podpiętym głównym skryptem biblioteki – *Lzwp* – oraz z komponentem *NetworkView*, który ten skrypt obserwuje (ma go w polu *Observed*). Obiekt ten można dodać do sceny wykorzystując prefabrykat *[LZWPlib]* z katalogu *Assets\LZWPlib\Prefabs*.

Skrypt Lzwp odpowiada za inicjalizację biblioteki. Podczas inicjalizacji zostaje on przeniesiony na pierwszy poziom hierarchii (jeśli nie był) i zostaje na nim wywołana metoda *DontDestroyOnLoad* [1], dzięki czemu nie zostanie usunięty przy zmianie lub przeładowaniu sceny. Obiekt ten ma charakter singletona – nadmiarowe jego instancje zostaną automatycznie usunięte.

Domyślnie inicjalizacja odbywa się w metodzie *Awake*. Można to zmienić poprzez pole *Init Mode*: na metodę *OnEnable*, *Start*, lub na inicjalizację manualną. W przypadku inicjalizacji manualnej przed rozpoczęciem korzystania z dowolnej funkcji biblioteki należy wywołać statyczną metodę *Init*:

```
Lzwp.Init();
```

Metodę tę można wywoływać wielokrotnie – jeśli biblioteka została już zainicjalizowana, kolejne wywołania metody zostaną automatycznie zablokowane.

Aby sprawdzić, czy biblioteka została zainicjalizowana, można posłużyć się statyczną właściwością *initialized*:

```
if (Lzwp.initialized)
{
    // biblioteka jest zainicjalizowana
}

if (Lzwp.initializinig)
{
    // biblioteka jest w trakcie inicjalizacji
}
```

Aby wykonać jakąś akcję tuż po inicjalizacji biblioteki, można skorzystać z właściwości *OnLZWPlibInitialize*. Zaleca się jednak użycie metod *AddAfterInitializedAction(Action action)* oraz *RemoveAfterInitializedAction(Action action)* odpowiednio do dodania lub usunięcia takiej akcji:

```
void OnEnable()
{
    Lzwp.AddAfterInitializedAction(Init);
}

void OnDisable()
{
    Lzwp.RemoveAfterInitializedAction(Init);
}


void Init()
{
    // ...
}
```

Jeśli podczas dodawania akcji wskazaną metodą biblioteka jest już zainicjalizowana, akcja ta zostanie wywołana natychmiastowo.

Podczas inicjalizacji ładowane są kolejno moduły biblioteki odpowiedzialne za: konfigurację, debugowanie (głównie logowanie), wyświetlanie, synchronizację, obsługę danych wejściowych. Są to kolejno instancje klas: *LzwpConfig*, *LzwpDebug*, *LzwpDisplay*, *LzwpSync*, *LzwpInput*. Można się później do nich odwołać poprzez statyczne właściwości klasy *Lzwp*: *config, debug, display, sync, input*.

Przy inicjalizacji ustalane jest również ID bieżącej instancji aplikacji (wartość numeryczna), które sprawdzić można poprzez statyczną właściwość *instanceID*:

```
int x = Lzwp.instanceID;
```

ID ustala się w następujący sposób:

- jeśli aplikacja uruchomiona została w edytorze, ID przyjmuje wartość ustawioną w polu *Editor Instance ID* w oknie edytora *LZWPlib* w zakładce *Testing*;

- jeśli aplikację uruchomiono poza edytorem, ID ustawiane jest na wartość przekazaną jako argument *instanceID* w linii poleceń (np. Aplikacja.exe -instanceID 3);
- jeśli brak argumentu linii poleceń, ID ustawiane jest na wartość będącą zawartością pliku *C:\instance-id.txt* (lub *D:\instance-id.txt*, jeśli ten wcześniejszy nie istnieje);
- jeśli nie udało się ustalić ID, przyjmowana jest wartość domyślna, czyli 0.

Moduł główny pozwala również sprawdzić wersję używanego pakietu LZWPlib oraz to, czy załadowany jest mock (namiastka), czy pełna jego wersja:

```
string version = Lzwp.GetVersion();  // "2.9"
bool mock = Lzwp.IsMock();
```

Udostępniona jest również metoda pozwalająca załadować urządzenie XR [2] [3] (w środowisku Unity skrót ten określa zbiorczo technologie VR, MR i AR) i odblokować jego użycie. Jest ona wywoływana przez bibliotekę podczas inicjalizacji. Przykład użycia:

```
Lzwp.Instance.LoadDevice("split");
```

Podczas inicjalizaji do logu aplikacji wypisane zostają następujące informacje:

- wersja biblioteki (oraz informacja czy jest to mock – namiastka);
- wersja silnika Unity;
- nadane przez Unity: wersja, GUID, nazwa (*Product name*) i autor (*Company name*) aplikacji;
- argumenty uruchomieniowe (np. przekazane w wierszu poleceń);
- nazwa i ścieżka ładowanej na starcie sceny;
- ID instancji;
- wpisy kolejno ładowanych modułów;
- informacja o zakończeniu inicjalizacji i czasie jej trwania.

1.7. *Logi*

Moduł debugowania (*Lzwp.debug*) obecnie udostępnia jedynie funkcje pozwalające pisać do logu aplikacji. Obudowują one metody z klasy *Debug* dostarczanej przez Unity, dołączając do każdego wpisu czas jego dodania (dla wersji silnika wcześniejszych niż 2018.1, w których tej informacji brakowało).

Dostępne są metody trzech typów: *Log* (informacja), *Warning* (ostrzeżenie) i *Error* (błąd). Jako argument przyjmują dowolny obiekt lub ciąg znaków z formatowaniem (opcjonalne parametry jako kolejne argumenty). Każdy z tych trzech typów posiada również metodę, w której jako pierwszy argument można przekazać kontekst: *LogWithContext*, *WarningWithContext* i *ErrorWithContext*. Dodatkowo są też dwie metody dla wyjątków: *Exception* (z wyjątkiem jako argument) i *ExceptionWithContext* (argumenty: kontekst, wyjątek). Przykładowe użycie:

```
Lzwp.debug.Log("Some <b>rich</b> text");
Lzwp.debug.Log("PI = {0}  TAU = {1}", Mathf.PI, 2f * Mathf.PI);
```

```
Lzwp.debug.Log(Vector3.right);
Lzwp.debug.LogWithContext(this, "Log with context");
Lzwp.debug.Warning("Unsupported type: {0}", typeof(SomeClass));
Lzwp.debug.Error("Error!");

try
{
    throw new Exception("Some exception");
}
catch (Exception ex)
{
    Lzwp.debug.Exception(ex);
    Lzwp.debug.ExceptionWithContext(this, ex);
}
```

Moduł ten pozwala także tworzyć logi w formacie stron HTML, w których kolejne pozycje są wyraźnie oddzielone i opatrzone kolorem tła zależnym od typu wpisu. Domyślnie logowanie takie jest wyłączone. Jego stan można zmienić poprzez plik konfiguracyjny (parametry *debug.logToHtml* i *debug.htmlLogFileName*; możliwość indywidualnego ustawienia dla każdej instancji) oraz poprzez argumenty linii poleceń (nadpisujące wartości z pliku konfiguracyjnego), np.:

- Aplikacja.exe -logToHtml *X* – włączenie logowania, gdy *X* przyjmuje wartość *0* lub *false*; wyłączenie logowania, gdy *X* przyjmuje inną wartość lub jest pominięty
- Aplikacja.exe -htmlLogFile "logfile.html" – włączenie logowania i ustawienie pliku logu.

Domyślna nazwa pliku logu w formacie HTML to *log_[INSTANCE_ID]_[DATETIME].html*. W nazwie pliku podmienione zostają:

- *[INSTANCE_ID]* – na numer bieżącej instancji aplikacji;
- *[DATETIME]* – na datę i czas utworzenia pliku logu (w formacie yyyy.MM.dd_HH.mm.ss.ffff).

1.8. *Konfiguracja aplikacji*

Aplikacja przygotowana z użyciem pakietu LZWPlib powinna być możliwa do uruchomienia w różnych środowiskach, m.in. w małej, średniej i dużej jaskini LZWP. Powinna również działać po zmianie konfiguracji danego środowiska oraz w środowiskach, które mogą powstać w przyszłości. Z tego powodu istotne ustawienia aplikacji pobierane są przy jej starcie z pliku konfiguracyjnego *LzwpStartupConfig.json*. Poza zestawem ustawień wymaganych przez poszczególne moduły biblioteki, plik ten może również zawierać ustawienia dodatkowe, zdefiniowane przez twórcę konkretnej aplikacji. Jeśli dany parametr konfiguracyjny nie zostanie zawarty w pliku (a w szczególności: jeśli plik nie zostanie odnaleziony), przyjmie on wartość domyślną, zdefiniowaną w kodzie biblioteki/aplikacji.

Dostęp do poszczególnych opcji konfiguracyjnych możliwy jest poprzez statyczne pole *Lzwp.config*, np.:

```
int numberOfScreens = Lzwp.config.screens.Length;

LzwpCameraConfig camCfg = Lzwp.config.instances[Lzwp.instanceID].cameras[0];
```

```
bool asymFrustum = camCfg.asymmetricFrustum;

if (cfg.cameraInstanceSetRenderingPath)
    GetComponent<Camera>().renderingPath = cfg.cameraInstanceRenderingPath;
```

Objaśnienie poszczególnych opcji znajduje się w opisie odpowiadających im modułów biblioteki.

Dla każdej aplikacji zdefiniować można dodatkowe opcje konfiguracyjne, które są niezależne od środowiska, w jakim zostanie ona uruchomiona. Można w ten sposób definiować np. prędkość poruszania się jakiegoś obiektu, liczbę obiektów do stworzenia czy też poziom trudności rozgrywki. Pakiet LZWPlib zawiera przykładowy, edytowalny skrypt pozwalający na przemieszczanie się po scenie (*MovementController*). Domyślnie dodatkowe ustawienia zawierają pole *movement* typu *MovementConfig*, przechowujące opcje konfiguracyjne wykorzystywane przez ten skrypt.

Aby dodać nowe pola do konfiguracji dodatkowej, należy dodać je do klasy cząstkowej *CustomAppConfig* (najlepiej w skrypcie, którego te ustawienia dotyczą), np.:

```
[Serializable]
public class MovementConfig
{
    public bool overwriteSceneMovementType = false;
    public bool overwriteOtherSceneMovementSettings = false;

    public MovementController.MovementConfig movementSettings =
      new MovementController.MovementConfig();
}

public partial class CustomAppConfig
{
    public MovementConfig movement = new MovementConfig();
}
```

Dla pól typu *Vector2, Vector2Int, Vector3, Vector3Int, Vector4, Quaternion, Rect* oraz *AnimationCurve* wskazać należy odpowiedni konwerter z użyciem atrybutu *JsonConverter*. LZWPlib dostarcza implementacje odpowiednich konwerterów w klasie *LzwpUtils.Converters*: *Vector2Converter, Vector2IntConverter, Vector3Converter, Vector3IntConverter, Vector3IntConverter, QuaternionConverter, RectConverter, AnimationCurveConverter*.

W przypadku kwaternionów zalecane jest użycie alternatywnego konwertera – *QuaternionEulerAnglesConverter* – który zamienia je na łatwiej edytowalną postać 3 kątów Eulera. Przykład pola z konwerterem:

```
[JsonConverter(typeof(LzwpUtils.Converters.Vector3Converter))]
public Vector3 axis = Vector3.one;
```

Odczyt dodatkowej konfiguracji odbywa się poprzez metodę (rozszerzenie) *GetCustom*, zwracającą obiekt klasy *CustomAppConfig*, np.:

```
bool overwriteMovementType =
      Lzwp.config.GetCustom().movement.overwriteSceneMovementType;
MovementConfig config = Lzwp.config.GetCustom().movement.movementSettings;
```

W projekcie aplikacji konfiguracja przechowywana jest w plikach typu *ScriptableObject*. W przypadku konfiguracji głównej są to obiekty klasy *AppConfigScriptableObject*, znajdujące się w katalogu *Assets\LZWPlib\Editor\Resources\Config*, natomiast w przypadku konfiguracji dodatkowej są to obiekty klasy *CustomAppConfigScriptableObject*, znajdujące się w katalogu *Assets\LZWPlib\Editor\Resources\CustomConfig*. Zawierają one pole *wrapper*, przechowujące właściwą konfigurację, będące typu odpowiednio *LzwpConfig* lub *CustomAppConfig*.

Obiekt konfiguracji można utworzyć na kilka sposobów:

- korzystając z menu głównego: *LZWPlib – Add config file / Add custom config file*;
- korzystając z menu kontekstowego w okienku *Project*: *Create – LZWPlib – Config file / Custom config file*;
- duplikując istniejący obiekt (zaznaczając go i używając kombinacji klawiszy Ctrl + D).

Pakiet LZWPlib zawiera kilka predefiniowanych obiektów konfiguracyjnych, które wykorzystać można podczas testowania aplikacji:

- konfiguracja główna:
  - *BigCAVE* / *MidiCAVE* / *miniCAVE* – konfiguracje używane w środowisku dużej / średniej / małej jaskini LZWP;
  - *BigCAVE_editor* / *MidiCAVE_editor* / *miniCAVE_editor* – konfiguracje analogiczne do powyższych, jednak przeznaczone do uruchamiania poza środowiskiem jaskini – np. w edytorze podczas testów; adres IP instancji głównej (serwerowej) jest w nich ustawiony na wartość 127.0.0.1, wyłączone jest uruchamianie systemu śledzenia;
  - *LZWP_Dev_PC* – konfiguracja dla stanowiska deweloperskiego LZWP;
  - *home* – konfiguracja dla testowania aplikacji poza LZWP (domyślna);
- konfiguracja dodatkowa:
  - *AppCustomConfig* – domyślne ustawienia poruszania się (latania i chodzenia).

Konfiguracje z obiektów ScriptableObject serializowane są do plików JSON za pomocą frameworka Json.NET załączonego do pakietu LZWPlib. Podczas budowania aplikacji w jej katalogu głównym tworzony jest plik *LzwpStartupConfig.json*, z którego konfiguracja ładowana jest przy starcie utworzonego programu. Również w przypadku uruchamiania aplikacji w edytorze ustawienia konfiguracyjne są serializowane (do pliku *Temp\LzwpStartupConfig.json*) a następnie deserializowane przy inicjalizacji biblioteki. W załącznikach nr 1 – 4 przedstawiona jest konfiguracja opracowana dla jaskiń i stanowisk deweloperskich znajdujących się w LZWP.

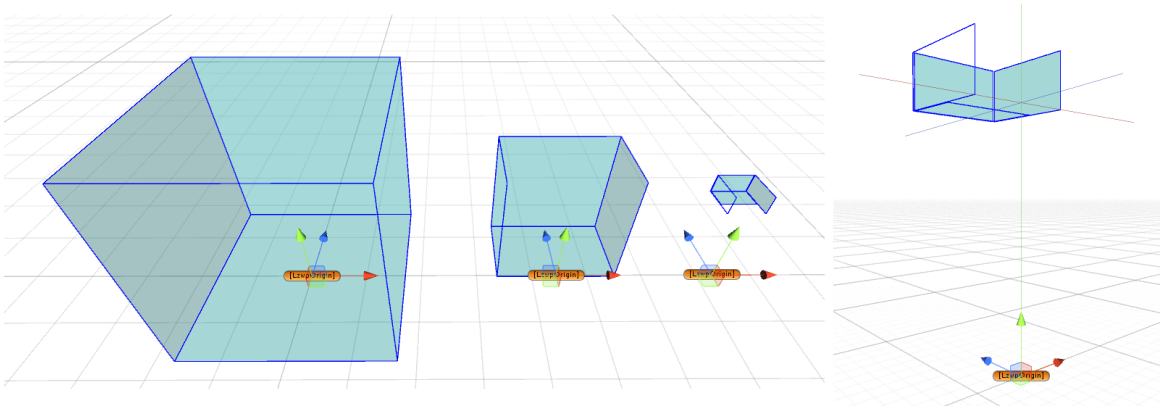
Czasami przydatny może się okazać prefabrykat *[EditConfigInPlayMode], który* zawiera skrypt *EditorOnly_EditConfigInPlayModeEditor* umożliwiający podgląd konfiguracji wczytanej przez aplikację uruchomioną w edytorze oraz modyfikację parametrów tej konfiguracji podczas działania aplikacji. Skrypt ten działa jedynie wewnątrz edytora i nie jest załączany do docelowej aplikacji podczas jej budowania.

1.9. *Reprezentacja jaskini i poruszanie się*

Każda scena w aplikacji powinna zawierać pojedynczy obiekt reprezentujący położenie i orientację rzeczywistej jaskini w wirtualnym świecie aplikacji. Reprezentacją taką jest obiekt, do którego przypięty jest skrypt *LzwpOrigin*. Przyjęto, że położenie tego obiektu wyznacza punkt znajdujący się na podłodze danego stanowiska (rys. 1.3):

- w BigCAVE'ie jest to środek podłogi;
- w MidiCAVE'ie jest to środek krawędzi ekranu podłogowego, będącej wejściem do jaskini;
- w MiniCAVE'ie jest to punkt rzutowany na fizyczną podłogę, nad którą zawieszone są monitory, ze środka odcinka łączącego przednie dolne wierzchołki jaskini (lewy dolny róg lewego monitora i prawy dolny róg prawego monitora).

Punkt ten jest początkiem lewoskrętnego układu współrzędnych jaskini, względem którego określana jest pozycja i rotacja obiektów takich jak ekrany jaskini czy też obiekty śledzone.



Rys. 1.3. Środek układu współrzędnych w poszczególnych jaskiniach LZWP

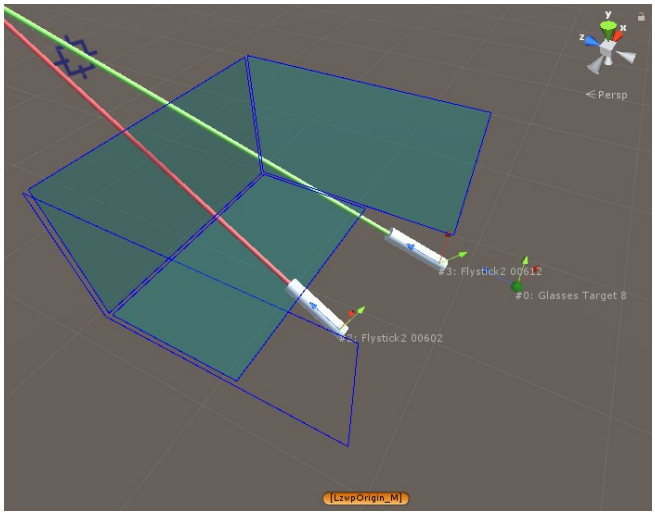
Aby dodać do sceny reprezentację jaskini, można posłużyć się jednym z dwóch dostępnych prefabrykatów: *[LzwpOrigin]* lub *[LzwpOrigin_M]*. Oba zawierają: wymagany skrypt; komponent *NetworkView*, służący synchronizacji położenia jaskini pomiędzy wszystkimi instancjami aplikacji; a także skrypt *LzwpOriginGizmos*, który pozwala wyświetlić graficzną reprezentację jaskini w oknie *Scene* edytora. Pierwszy z prefabrykatów jest zestawem minimalnym, drugi zaś zawiera dodatkowo skrypt *MovementController* – umożliwiający przemieszczanie jaskini po świecie wirtualnym – oraz wymagane przez ten skrypt komponenty (*Rigidbody*, *Audio Source*, *Capsule Collider*, *Character Controller*).

Skrypt *LzwpOrigin* zawiera pole *Obstacle Walls Material*, do którego można podpiąć materiał używany do wyświetlania ostrzeżeń o zbliżaniu się do przeszkody fizycznej (np. ściany jaskini).

Skrypt ten umożliwia pobranie pozycji i rotacji jaskini poprzez statyczne metody *GetPosition* i *GetRotation*, np.:

```
Vector3 CavePosition = LzwpOrigin.GetPosition();
```

Skrypt *LzwpOriginGizmos* umożliwia wyświetlenie w oknie sceny reprezentacji ekranów wybranej jaskini, płaszczyzn przeszkód fizycznych oraz reprezentacji śledzonych obiektów (rys. 1.4). W trybie edycji z rozwijanej listy obiektów konfiguracyjnych, znajdującej się na górze tego komponentu, wybrać można, która jaskinia będzie reprezentowana (domyślnie będzie to ta, dla której obiekt konfiguracyjny wskazano w ustawieniach projektu). W trybie gry natomiast, czyli gdy aplikacja uruchomiona jest w edytorze, reprezentacja jaskini będzie opierała się na wczytanym podczas inicjalizacji biblioteki pliku konfiguracyjnym. Możliwe jest dostosowanie wyświetlanych reprezentacji obiektów: zmiana widoczności, kolorów, rysowania nazw, rysownia lokalnych układów współrzędnych itd. Komponent skryptu musi być rozwinięty w oknie inspektora, aby obiekty mogły być rysowane w oknie sceny.



Rys. 1.4. Ekrany małej jaskini oraz 3 śledzone obiekty prezentowane w oknie sceny

Skrypt *MovementController*, załączony do drugiego z prefabrykatów, umożliwia poruszanie się po scenie za pomocą myszy, klawiatury, kontrolerów Flystick (dowolnego z dostępnych lub konkretnie wskazanego) oraz sferycznego symulatora chodu. Udostępnia dwa tryby przemieszczania się: chodzenie oraz latanie. Skrypt ten jest edytowalny, można więc zmodyfikować go według własnych potrzeb. Posiada również wiele opcji konfiguracyjnych, które są przykładem opisanej wcześniej konfiguracji dodatkowej. Wartości domyślne ustawić można w komponencie tego skryptu w oknie inspektora.

Obsługa chodzenia opiera się na mechanizmach udostępnianych w paczce standardowych zasobów Unity (ang. *Unity Standard Assets*). Może wykorzystywać komponent *Character Controller* (domyślnie) lub bazować na fizyce. Zgodnie z domyślną konfiguracją chodzić można z użyciem klawiszy W, S, A i D na klawiaturze. Klawisz Ctrl spowalnia ruch, Shift umożliwia bieganie, Spacja – skok. Przytrzymując wciśnięty prawy przycisk myszy można się z jej użyciem rozglądać dookoła.

Za pomocą joysticka, znajdującego się na górze kontrolera Flystick, można się obracać, wychylając gałkę w lewo lub w prawo, oraz przemieszczać do przodu lub do tyłu w kierunku wskazywanym przez kontroler, wychylając gałkę odpowiednio w przód lub w tył. Prędkość zależna jest od stopnia wychylenia gałki. Spust kontrolera umożliwia skok. W trybie chodzenia wykorzystywany jest obiekt kolizji (*Capsule Collider* lub pochodzący z komponentu *Character Controller*), który nie przemieszcza się za fizycznie chodzącym użytkownikiem i zawsze stoi w punkcie początkowym układu współrzędnych jaskini. Podczas chodzenia odgrywane są dźwięki kroków, puszczany jest też dźwięk wyskoku i lądowania.

W trybie latania kolizje są wyłączone. Podobnie jak w przypadku chodzenia – klawisze W i S umożliwiają lot do przodu i do tyłu względem kierunku patrzenia, klawisze A i D – lot na boki, dodatkowo natomiast klawisze Q i E – lot do góry i w dół. Klawisz spacji również powoduje lot w górę. Shift zwiększa prędkość lotu, Ctrl – zmniejsza. Rozglądać się można z użyciem myszki, przytrzymując jej prawy przycisk. Z użyciem kontrolera Flystick latać można w przód i w tył w kierunku przez niego wskazywanym, wychylanie gałki joysticka na boki powoduje obrót.

Sferyczny symulator chodu umożliwia ruch do przodu, do tyłu i na boki w obu omówionych trybach.

Podczas działania aplikacji można przełączać się pomiędzy trybem chodzenia i latania za pomocą szybkiego, dwukrotnego wciśnięcia klawisza skoku (Spacja / spust kontrolera).

1.10. *Wyświetlanie*

Zazwyczaj przy renderowaniu grafiki bryła widzenia jest ostrosłupem (mamy do czynienia z projekcją perspektywiczną) i jest ustawiana symetrycznie. Zakłada to, że widz patrzy na ekran z pewnej ustalonej odległości, a jego oczy są na wprost środka ekranu. Wirtualne rozglądanie się polega wówczas na obracaniu takiej bryły (względem czubka ostrosłupa) przy jednoczesnym zachowaniu jej kształtu. Ma to zastosowanie w przypadku korzystania ze standardowego monitora czy też wielkiego ekranu (tak jak podczas projekcji na audytorium przyległym do LZWP), gdzie punkt, z którego patrzy użytkownik, nie jest śledzony. Podobnie jest też w przypadku gogli rzeczywistości wirtualnej – pozycja oczu względem ekranu nie zmienia się, chociaż głowa użytkownika jest śledzona i może się przemieszczać.

W przypadku jaskiń rzeczywistości wirtualnej sprawa wygląda nieco inaczej. Przyjmijmy, że jaskinia taka jest rodzajem pokoju, a jej ekrany są oknami, przez które użytkownik ogląda świat na zewnątrz jaskini (to samo dotyczyć będzie oglądania obiektów wdzierających się przez okno do wnętrza). Załóżmy również, że użytkownik zamknął jedno oko i ogląda świat w trybie monoskopowym. Samo rozglądanie się, polegające na obracaniu gałki ocznej względem pokoju, nie zmienia w żaden sposób tego, co widać przez okno – obraz „na szybie” jest taki sam. Dopiero zmiana pozycji głowy – zmiana punktu, z którego dokonujemy obserwacji – wpływa na to, że widok za oknem zmienia się. Poruszając się w płaszczyźnie równoległej do okna część obrazu chowa się za jego krawędzią, nowa część natomiast ukazuje zza krawędzi przeciwległej. Podchodząc bliżej okna widzimy coraz więcej obrazu wyłaniającego się zza wszystkich krawędzi jednocześnie, odchodząc – widzimy coraz mniej.

Można więc zauważyć, że ostrosłup widzenia, mający swój czubek w miejscu oka, wypełnia obszar okna tak, że krawędzie boczne tego ostrosłupa przechodzą przez narożniki okna (zakładając, że jest to typowe okno prostokątne). Taka bryła widzenia jest asymetryczna.

W silnikach graficznych uzyskanie odpowiedniego ustawienia bryły widzenia polega na właściwym dobraniu wartości w macierzy widoku oraz macierzy projekcji kamery. Silnik gier Unity daje możliwość bezpośredniej modyfikacji tych macierzy. Znając fizyczne położenie ekranów i ich rozmiary (dzięki danym konfiguracyjnym), a także punkt, z którego patrzy użytkownik (dzięki danym z systemu śledzenia), jesteśmy w stanie wyznaczyć odpowiednie wartości. Ustawienie macierzy dla symetrycznej bryły widzenia następuje tylko raz, na początku działania aplikacji, zaś dla bryły asymetrycznej – występuje w każdej klatce przed rozpoczęciem procesu wycinania elementów niewidocznych (ang. *viewing frustum culling*), czyli w metodzie *OnPreCull*.

Konfiguracja startowa aplikacji zawiera liczne parametry dotyczące wyświetlania obrazu, dzięki czemu możliwe jest działanie aplikacji w różnych konfiguracjach jaskiń.

Obiekt konfiguracyjny *Display* zawiera ogólne parametry wyświetlania:

- *Stereoscopy Mode* – tryb stereoskopii: tryb używany w jaskiniach (*Stereo*), podzielony ekran (*Split*) lub brak stereoskopii (*None*);
- *Eye separation* – rozstaw oczu (w metrach);
- *Swap Eyes* – zamiana obrazu dla oka lewego z obrazem dla oka prawego;
- *Fulsscreen* – tryb pełnoekranowy;
- *Set Window Pos* – określa czy ustawić pozycję okna, jeśli aplikacja nie działa w trybie pełnoekranowym;
- *Window Pos* – docelowa pozycja okna, jeśli poprzedni parametr został włączony;
- *Cursor Visible* – określa czy w aplikacji widoczny ma być kursor myszki.

Parametr konfiguracyjny *Screens* jest listą dostępnych w ramach danej jaskini fizycznych ekranów. Każdy z nich ma przypisaną własną nazwę i zdefiniowany jest następującymi parametrami:

- *Center* – współrzędne środka ekranu względem początku układu współrzędnych jaskini;
- *Normal* – wektor normalny, idący od ekranu w kierunku widza, w odniesieniu do układu współrzędnych jaskini;
- *Up* – wektor wskazujący kierunek górny, czyli kierunek, w którym znajduje się górna krawędź ekranu, w odniesieniu do układu współrzędnych jaskini;
- *Width* – szerokość ekranu (w metrach);
- *Height* – wysokość ekranu (w metrach).

Taki sam zestaw parametrów służy do opisania płaszczyzn przeszkód fizycznych (zwykle są to ściany jaskini), na których będzie wyświetlane ostrzeżenie o zbliżaniu się do nich śledzonych obiektów. Tablica tych obiektów zawarta jest w parametrze konfiguracyjnym *Obstacle Walls*. Pozycja zarówno ekranów jak i płaszczyzn przeszkód liczona jest z uwzględnieniem przesunięcia zdefiniowanego w parametrze *Screens And Obstacle Walls Offset From Origin*.

LZWPlib umożliwia renderowanie obrazu z różnych punktów widzenia (ang. *points of view*). Punkt widzenia powiązany jest ze śledzonym obiektem (tzw. pozą, ang. *pose*), za którym podąża – zwykle są to okulary. Śledzoną pozę można zmieniać w trakcie działania aplikacji, dzięki czemu można np. płynnie lub natychmiastowo przełączać dopasowanie obrazu do perspektywy kolejnych użytkowników ze śledzonymi okularami – zamiast fizycznie wymieniać się okularami wiodącymi. Można zdefiniować wiele punktów widzenia, co może się okazać przydatne np. w aplikacjach z multistereoskopią. Parametr konfiguracyjny *Points Of View* jest listą tego typu punktów, z których każdy przypisany ma indeks śledzonej przez niego początkowo pozy, odnoszący się do listy *Lzwp.input.poses*. Po inicjalizacji biblioteki tablica *Lzwp.display.pointsOfView* zawiera zdefiniowane w konfiguracji punkty widzenia. Pozycję i orientację każdego z nich można pobrać korzystając z właściwości *position* oraz *rotation*. Pozę (bieżącą lub tę, do której trwa właśnie płynne przejście) odczytać można za pośrednictwem właściwości *targetPose*, natomiast do ustawienia innej wykorzystać należy metodę *SetPose*, w której jako argument przekazać można nową pozę lub jej indeks w tablicy *Lzwp.input.poses*, a także opcjonalnie czas płynnego przejścia do niej (w sekundach). Przykładowo przypisanie z półsekundowym płynnym przejściem pozy okularów drugich jako głównego punktu widzenia (z indeksem 0) może wyglądać następująco:

```
Lzwp.display.pointsOfView[0].SetPose(Lzwp.input.glasses[1], 0.5f);
```

Parametr konfiguracyjny *Instances* definiuje listę możliwych instancji aplikacji. Każda uruchomiona instancja programu odnosi się do jednej z pozycji tej listy. Każda pozycja zawiera listę kamer (parametr *Cameras*) oraz obiekt *Display*, w którym można nadpisać dla danej instancji wartości poszczególych opisanych wcześniej parametrów tego typu obiektu. Każda kamera zawiera natomiast następujące parametry:

- *Screen* – indeks ekranu z omówionej wcześniej listy ekranów;
- *Display Idx* – indeks ekranu z systemowej listy podłączonych ekranów (np. monitorów, projektorów); Unity może obsłużyć do 8 takich ekranów jednocześnie [4];
- *Viewport* – określa położenie i wymiary obszaru w wynikowym obrazie, za którego wypełnienie odpowiada dana kamera;
- *Asymmetric Frustum* – czy bryła widzenia ma być asymetryczna, czyli czy obraz będzie dopasowany do ekranu, czy też będzie podążał za kierunkiem patrzenia użytkownika;
- *Convergence Distance* – dotyczy wyłącznie symetrycznej bryły widzenia i określa odległość płaszczyzny, w której obraz dla lewego i prawego oka pokrywa się (odległość ekranu od oczu użytkownika);
- *Point Of View Idx* – indeks pozy z omówionej wcześniej listy punktów widzenia;
- *Stereoscopy workarounds* – parametry nadpisujące globalne ustawienia obejść problemów ze stereoskopią.

Podczas inicjalizacji modułu wyświetlania następuje ustawienie pozycji okna (jeśli jest to zdefiniowane w konfiguracji), zainicjalizowanie odpowiedniego trybu stereoskopii oraz utworzenie na podstawie „wzoru kamery” odpowiedniej liczby kamer i zainicjalizowanie ich.

„Wzór kamery” to zestaw obiektów danej sceny oznaczonych tagiem *MainCamera*, które nie zostały wyłączone w oknie inspektora, i do których przypięty jest skrypt *LzwpCamera*. Jeśli żaden obiekt na scenie nie jest oznaczony tagiem *MainCamera*, warunek ten jest pomijany. Jeśli „wzór” nie zostanie znaleziony, inicjalizacja kamer zostaje przerwana. LZWPlib nie będzie zajmował się wówczas dopasowywaniem obrazu. Jeśli jednak „wzór” zostanie znaleziony, jest on duplikowany odpowiednią liczbę razy (zależną od liczby kamer w konfiguracji danej instancji).

„Wzór kamery” można stworzyć samemu, odpowiednio modyfikując istniejące już na scenie obiekty kamer (ustawienie tagu, dodanie skryptu *LzwpCamera*). Można również posłużyć się prefabrykatem *[CameraContainer]*, który jest właśnie tego typu „wzorem” i zawiera w sobie dwa obiekty z kamerami, z których jedna przeznaczona jest dla oka lewego, a druga dla prawego. Możliwe jest również użycie pojedynczego obiektu kamery, renderującego stereoskopowy obraz dla obu oczu (o ile wyświetlanie stereoskopowe jest włączone) – parametr *Target Eye* kamery ustawiony na wartość *Both*.

Skrypt *LzwpCamera* odpowiada za odpowiednie ustawienie danej kamery tak, aby wyświetlała obraz dopasowany do odpowiadającego jej ekranu. W tym celu podczas inicjalizacji zmienia następujące parametry kamery: *Stereo Separation* (ustawia na 0, inne wartości powodują błędne wyświetlanie), *Viewport Rect, Target Display, Projection, Allow HDR, Allow MSAA*, opcjonalnie również *Rendering Path*. Z tego względu nie należy modyfikować tych parametrów samemu. Zalecane jest natomiast odpowiednie ustawienie płaszczyzn obcinania (parametr *Clipping Planes*). Po inicjalizacji skrypt na bieżąco aktualizuje pozycję kamery tak, oby odpowiadała pozycji oczu (śledzonych okularów), a następnie ustawia jej macierze projekcji i widoku. Nie należy zmieniać pozycji kamery samemu, gdyż i tak zostanie nadpisana pozycją otrzymaną od systemu śledzenia. Nie należy również umieszczać tych kamer w poruszających się obiektach, ponieważ może to skutkować drżeniem obrazu (spowodowanym aktualizacją położenia kamery na podstawie różnych źródeł).

Jeśli kamera znajdzie się za płaszczyzną ekranu (np. osoba w śledzonych okularach wyjdzie poza jaskinię), to przy wyznaczaniu macierzy wykorzystywany będzie wektor normalny ekranu o przeciwnym zwrocie.

Stereoskopia nie działa w przypadku uruchamiania aplikacji wewnątrz edytora (wyświetlany jest wówczas obraz dopasowany do jednego z oczu).

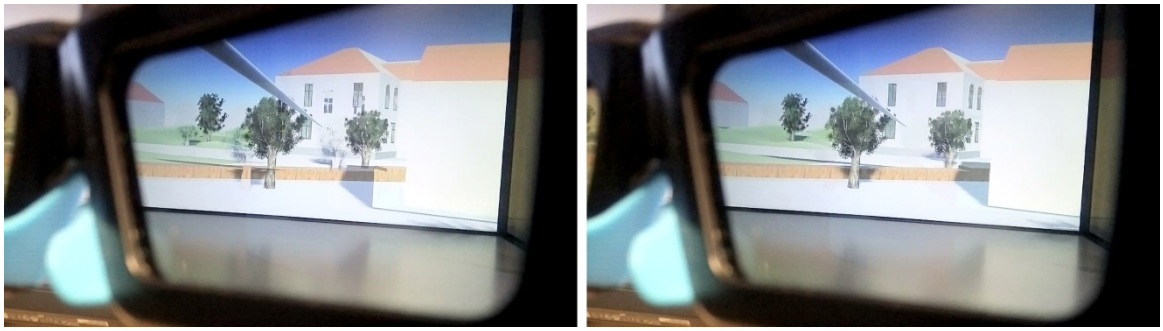
Unity natywnie obsługuje stereoskopię w DirectX od wersji 11.1 tego API [5] (nie jest obsługiwane poczwórne buforowanie wykorzystujące rozszerzenie opracowane przez AMD). Wersja DirectX 11.1 wymaga systemu operacyjnego Windows 8.1 lub nowszego. Ponieważ średnia i duża jaskinia LZWP używa wciąż starszego systemu operacyjnego (Windows 7), nie jest możliwe uzyskanie stereoskopii w ten sposób. W systemie Windows 7 (jak również nowszych) działa natomiast stereoskopia w aplikacjach uruchomionych w trybie OpenGL.

Wyświetlanie w trybie stereoskopowym czasami powoduje nieoczekiwane efekty i błędy obrazu. Najczęstszy problem objawia się „zawieszaniem” obrazu dla jednego z oczu i wyświetlaniem mu przez cały czas pierwszej wyrenderowanej klatki. Dzieje się to między innymi wtedy,

gdy w aplikacji wykorzystywany jest animowany skybox. Problem ten można obejść wyświetlając skybox na wewnętrznych ściankach wielkiej bryły (sześcianu lub sfery) „nałożonej" na scenę. Niepożądany efekt tego typu powodują również odbicia w standardowo dostępnej w Unity wodzie [6], dlatego należy je wyłączyć.

Stereoskopia nie działa zazwyczaj przy włączonych opcjach HDR i MSAA w komponencie kamery.

Jeśli do renderowania obrazu dla obojga oczu używana jest pojedyncza kamera, to czasami można zaobserwować problem z cieniami w klasycznym trybie renderowania (ang. *forward rendering*) – cienie odpowiednie dla jednego z oczu widoczne są w obrazach wynikowych obu (rys. 1.5). Problem znika po przełączeniu na renderowanie odroczone (ang. *deferred rendering*).



Rys. 1.5. Obraz dla prawego oka renderowany w trybie klasycznym (widoczne problemy z cieniami) i odroczonym (poprawne cienie)

Kolejny problem dotyczy przedwczesnego wycinania obiektów (lub cieni obiektów) znajdujących się blisko krawędzi ekranu. Pewnym obejściem tego problemu jest ustawienie szerokiego pola widzenia (na poziomie 179°) – parametr *Field of View*. Nie zepsuje to dopasowania obrazu, gdyż LZWPlib i tak nadpisuje macierze kamery, jednak może negatywnie wpłynąć na wydajność aplikacji, ponieważ znacząco zmniejsza liczbę elementów pomijanych podczas renderowania.

Elementy interfejsu użytkownika rysowane za pośrednictwem klas *GUI* i *GUILayout* widoczne będą tylko na obrazie dla lewego oka. Podobnie z elementami interfejsu umieszczanymi jako obiekty sceny (np. *Button*). W przypadku tej drugiej grupy rozwiązaniem może być ustawienie rysowania ich jako elementy świata (ang. *world space*), a nie jako rysowana na końcu „nakładka na ekran" (ang. *screen space overlay*), będąca opcją domyślną. Ustawienia tego można dokonać w nadrzędnym dla kontrolek UI (interfejs użytkownika, ang. *user interface*) obiekcie sceny typu *Canvas*, zmieniając wartość pola *Render Mode* na *World Space* w komponencie *Canvas*. Aby UI nie było przesłaniane przez inne obiekty sceny można np. utworzyć dla niego nowy materiał ze zmodyfikowanym domyślnym programem cieniującym. Program ten dla odpowiedniej wersji edytora można pobrać z archiwum Unity [7]. Jego kod zawarty jest w pliku *DefaultResourcesExtra\UI\UI-Default.shader*. Zmodyfikować wystarczy w nim jedną linijkę: *ZTest [unity_GUIZTestMode]* podmienić na *Ztest Always* (taką wartość przyjmuje parametr *[unity_GUIZTestMode]*, gdy tryb renderowania ustawiony jest na *Screen Space - Overlay*).

1.11. *Dźwięk*

Jeśli opisany w poprzednim podrozdziale „wzór kamery” zawierał komponent *Audio Listener*, to komponent ten zostaje podczas inicjalizacji usunięty i tworzony jest obiekt *[Ears]* („uszy”) zawierający nową instancję tego komponentu oraz skrypt *LzwpEars* (który odpowiada za ustawianie pozycji tego obiektu na środek głowy pierwszego śledzonego użytkownika).

Źródłem dźwięku w jaskiniach LZWP jest komputer będący węzłem głównym, na którym uruchamiane są aplikacje w trybie serwera. Nie ma więc potrzeby odtwarzania dźwięku w instancjach klienckich.

1.12. *System śledzenia i obsługa kontrolerów*

LZWPlib zapewnia natywne wsparcie dla systemów śledzenia firmy ART Advanced Realtime Tracking GmbH. Użycie dostarczonego przez tę firmę SDK napisanego w języku C++ wymaga przygotowania wtyczki natywnej dla silnika Unity – tak też wstępnie zrobiono przygotowując pakiet LZWPlib. Okazało się jednak, że chęć niezależnego połączenia z aplikacją ART-Human, służącą do wyznaczania pozy ciała (szkieletu) osoby ubranej w zestaw śledzenia ciała, niesie za sobą konieczność znacznych modyfikacji tej wtyczki. Postanowiono więc zarzucić jej wykorzystanie i zaimplementować własną obsługę w kodzie zarządzanym komunikacji z systemem śledzenia i przetwarzania odebranych od niego wyników.

Dla każdego systemu typu DTrack zdefiniowanego w konfiguracji (na liście *Input – Dtracks*) przy inicjalizacji biblioteki tworzony jest obiekt klasy *DTrack*, który dodawany jest do tablicy *Lzwp.input.dtracks*. Następnie, na podstawie list *Bodies*, *Flysticks* i *Hands* z konfiguracji danego systemu śledzenia, uzupełniane są listy *poses*, *glasses*, *flysticks* i *hands* modułu *Lzwp.input*. Lista *poses* zawiera obiekty typu *LzwpPose* odpowiadające wszystkim śledzonym obiektom. Lista *glasses* zawiera pozy obiektów śledzonych, które w konfiguracji oznaczone są jako okulary. Listy *flysticks* i *hands* natomiast zawierają odpowiednio obiekty typu *Flystick* lub *FingertrackingHand*, których pole *pose* odnosi się do właściwej im pozy z listy *poses*. Odpowiadają one kontrolerom Flystick oraz dłoniom śledzonym z wykorzystaniem systemu *Finger Tracking*.

Obiekt *LzwpPose* zawiera następujące pola i właściwości:

- *index* – indeks pozy na liście *Lzwp.poses*;
- *name* – nazwa pozy zawierająca jej indeks oraz nazwę zdefiniowaną w konfiguracji;
- *position* – globalna pozycja pozy na scenie (z uwzględnieniem pozycji i rotacji jaskini) – tylko do odczytu;
- *rotation* – globalna rotacja (kwaternion) pozy na scenie (z uwzględnieniem pozycji i rotacji jaskini) – tylko do odczytu;
- *positionLocal* – lokalna pozycja pozy względem jaskini;
- *rotationLocal* – lokalna rotacja pozy względem jaskini;
- *glasses* – określa, czy śledzony obiekt, któremu ta poza odpowiada, jest okularami;

- *prevTracked* – określa, czy poza była śledzona (śledzony obiekt był widoczny) przy poprzedniej aktualizacji stanu;
- *tracked* – określa, czy poza była śledzona (śledzony obiekt był widoczny) przy ostatniej aktualizacji stanu;
- *wasTracked* – określa, czy poza była śledzona (śledzony obiekt był widoczny) od momentu uruchomienia aplikacji;
- *OnTrackingStateChanged* – akcja wywoływana po zmianie stanu śledzenia pozy (stanu widoczności śledzonego obiektu); jako argument przyjmuje bieżący stan śledzenia (widoczności).

Początkowa pozycja i rotacja pozy ustawiane są na wartości parametrów konfiguracyjnych *Input - Poses Starting Position* i *Input - Poses Starting Rotation*. Konfiguracja zawiera również parametry pozwalające skorygować wartości pozycji i rotacji poszczególnych póz przesłane przez system śledzący (parametry *Position Correction* i *Rotation Correction*). Jest też parametr pozwalający określić położenie początku układu współrzędnych systemu śledzenia względem początku układu współrzędnych jaskini - *Offset From Origin*. Pozy są automatycznie synchronizowane pomiędzy wszystkimi instancjami aplikacji.

Obiekt *Flystick* zawiera następujące pola:

- *pose* – poza kontrolera (odnosi się do pozy z listy *Lzwp.input.poses*);
- *buttons* – 6-elementowa tablica przycisków – obiektów klasy *Button*;
- *joysticks* – 2-elementowa tablica wartości wychylenia gałki joysticka w osi poziomej (lewo/prawo) i pionowej (tył/przód); wartości te są w przedziale <-1, 1>;
- *OnButtonChange* – akcja wywoływana po zmianie stanu przycisku (wciśnięcie lub zwolnienie); jako argument przyjmuje identyfikator przycisku (*ButtonID*);
- *OnButtonPress*, *OnButtonRelease* – akcje wywoływane odpowiednio po wciśnięciu lub zwolnieniu przycisku; jako argument przyjmują identyfikator przycisku (*ButtonID*).

Zawiera również typ wyliczeniowy *ButtonID*, którego elementy odpowiadają przyciskom kontrolera:

- *Fire* – spust/cyngiel; wartość 0,
- *Button4* – przycisk czwarty od lewej; wartość 1;
- *Button3* – przycisk trzeci od lewej; wartość 2;
- *Button2* – przycisk drugi od lewej; wartość 3;
- *Button1* – przycisk pierwszy od lewej; wartość 4;
- *Joystick* – przycisk w joysticku; wartość 5;
- *Unknown* – nieokreślony przycisk; wartość 99.

Obiekt *Button* zawiera pola:

- *wasPressed* – określa, czy przy poprzedniej aktualizacji stanu przycisk został wciśnięty;
- *wasReleased* – określa, czy przy poprzedniej aktualizacji stanu przycisk został zwolniony;
- *isActive* – określa, czy przy poprzedniej aktualizacji stanu przycisk był wciśnięty;

- *OnChange* – akcja wywoływana po zmianie stanu przycisku (wciśnięcie lub zwolnienie);
- *OnPress* – akcja wywoływana po wciśnięciu przycisku;
- *OnRelease* – akcja wywoływana po zwolnieniu przycisku.

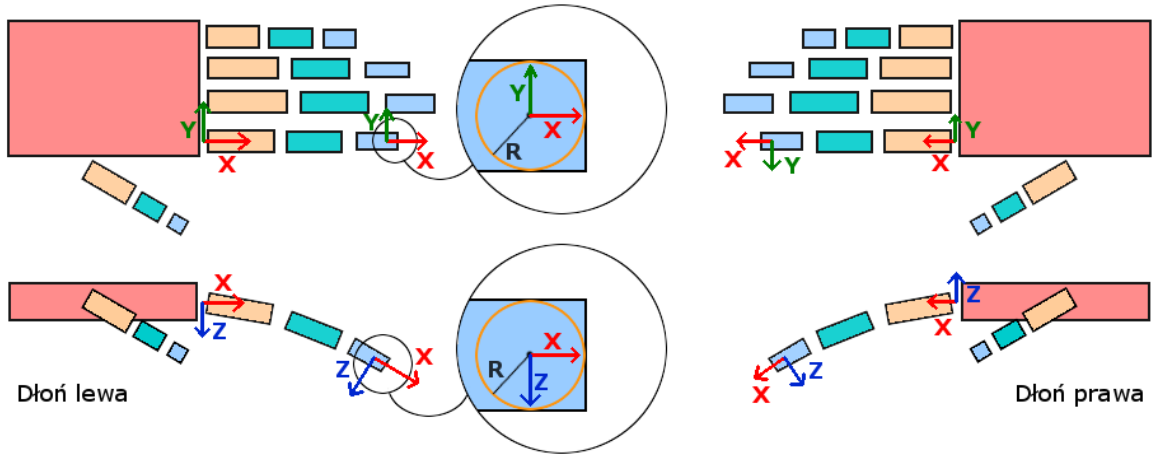
Obiekt *FingertrackingHand* zawiera pola:

- *pose* – poza dłoni (odnosi się do pozy z listy *Lzwp.input.poses*);
- *leftHand* – określa, czy jest to dłoń lewa;
- *rightHand* – określa, czy jest to dłoń prawa;
- *fingers* – 5-elementowa tablica obiektów *FingertrackingFinger*, odpowiadających kolejnym palcom dłoni, licząc od kciuka.

Obiekt *FingertrackingFinger* zawiera pola:

- *tipPose* – poza czubka palca (sfery wpisanej w ten czubek);
- *tipRadius* – promień sfery wpisanej w czubek palca (w metrach);
- *phalanxLength* – 3-elementowa tablica długości (w metrach) kolejnych paliczków danego palca (licząc od końca palca);
- *phalanxAngle* – 2-elementowa tablica kątów (w stopniach) odpowiadających rozwarciu kolejnych stawów danego palca (licząc od końca palca: stawu między 1. i 2. paliczkiem oraz stawu między 2. i 3. paliczkiem);
- *joints* – 4-elementowa tablica obiektów *FingertrackingFingerJoint*, odpowiadających czubkowi oraz trzem kolejnym stawom danego palca.

Obiekt *FingertrackingFingerJoint* zawiera trzy 3-elementowe wektory – *positionInFingerCoordSystem*, *positionInHandCoordSystem* i *positionInRoomCoordSystem* – odpowiadające pozycjom danego stawu odpowiednio w układzie współrzędnych palca, dłoni lub systemu śledzenia. Układy współrzędnych palców i dłoni przedstawione są na rys. 1.6.



Rys. 1.6. Układy współrzędnych palców i dłoni; *R* oznacza promień sfery wpisanej w czubek palca [16]

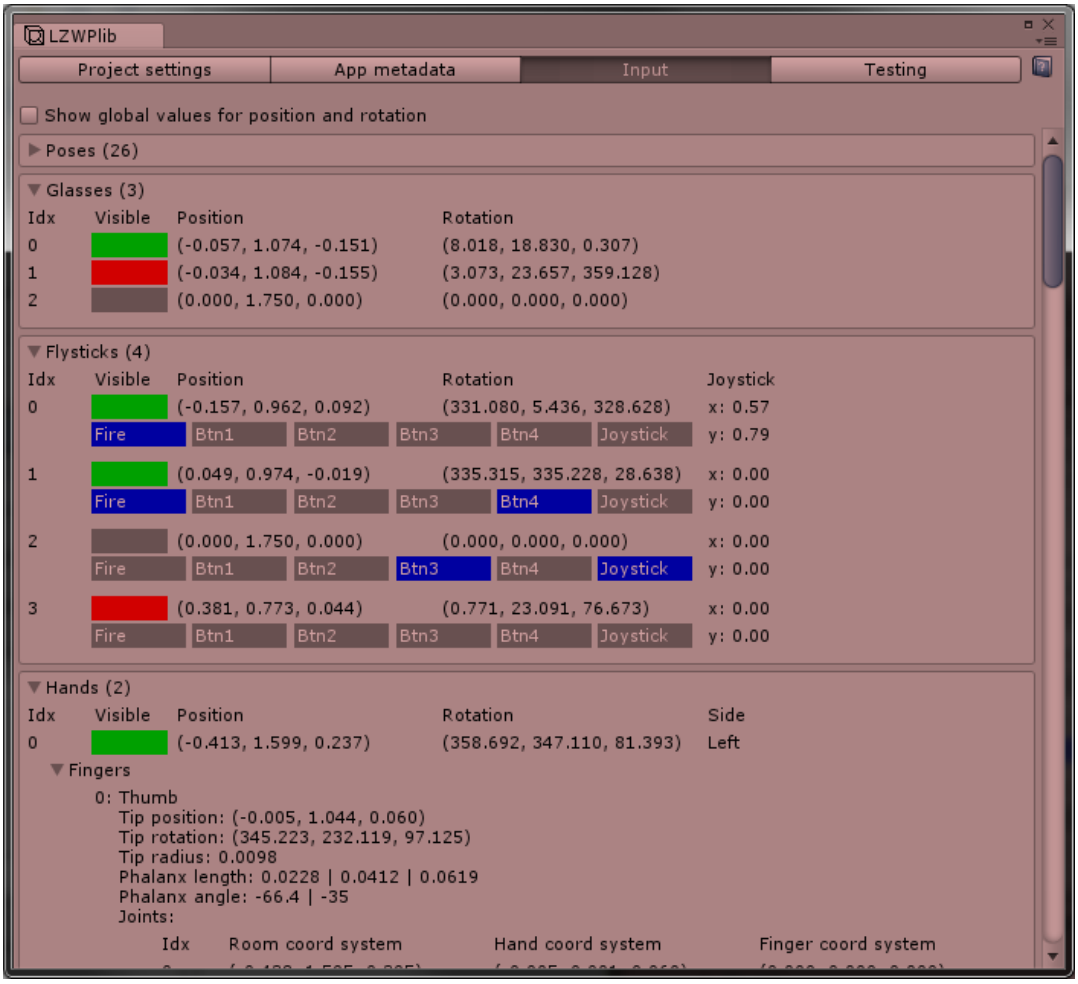
Po uzupełnieniu list *poses*, *glasses*, *flysticks* i *hands* na podstawie listy *Points Of View* z ustawień konfiguracyjnych uzupełniana nowymi obiektami jest też lista punktów widzenia (*Lzwp.display.pointsOfView*). Każdemu z nich przypisywana jest odpowiednia poza.

Następnie, jeśli w konfiguracji dla danego systemu śledzenia jest on odblokowany (parametr *Enabled*), następuje:

- jeśli ustawione jest rozpoczynanie śledzenia z poziomu aplikacji (parametr *Start Measurement*) – próba połączenia (TCP) z główną jednostką systemu śledzenia (kontrolerem) pod zdefiniowanym w konfiguracji adresem IP (parametr *Controller IP*) i domyślnym portem 50105. Jeśli uda się połączyć i aplikacja może kontrolować system (tylko jedno połączenie w danym momencie może mieć takie uprawnienia), wysyłana jest komenda rozpoczęcia śledzenia. Jeśli śledzenie było już uruchomione, jest ono najpierw zatrzymywane. Jeśli aplikacja nie może kontrolować systemu śledzenia, następuje sprawdzenie jakie inne połączenie blokuje ten dostęp. Do logu wypisywany jest otrzymany w odpowiedzi adres IP tego połączenia wraz z numerem portu, a jeśli się uda pobrać dodatkowe dane przy użyciu programu netstat – również nazwa i identyfikator blokującego procesu;
- rozpoczęcie asynchronicznego odbierania i przetwarzania danych – nie jest blokowany główny wątek aplikacji.

Aplikacja nasłuchuje na zdefiniowanym w konfiguracji porcie UDP (parametr *Data Port*, domyślnie: 5000). System śledzenia po każdym dokonanym pomiarze wysyła ramkę w tekstowym formacie ASCII, zawierającą w kolejnych linijkach parametry wskazane uprzednio w opcjach konfiguracyjnych tego systemu. Po odczytaniu i przetworzeniu danej ramki aktualizowane są pozy oraz pozostałe parametry (stan joysticków, przycisków), a także wywoływane odpowiednie akcje – w szczególności akcja *Lzwp.input.PosesUpdated*, oznajmiająca pojawienie się aktualnych danych.

Bieżące informacje o odczytanych wartościach dla poszczególnych póz, okularów, kontrolerów Flystick, dłoni i palców oraz kości szkieletu podejrzeć można w zakładce *Input* okna *LZWPlib* (rys. 1.7). Dane wyświetlane są wyłącznie wtedy, gdy aplikacja jest uruchomiona. Jeśli zaznaczono opcję *Show global values for position and rotation*, to dla pozycji i rotacji obiektów wyświetlane są wartości globalne (odnoszące się do układu współrzędnych wirtualnego świata), w przeciwnym wypadku – lokalne (odnoszące się do obiektu reprezentującego jaskinię). Szare pole w kolumnie *Visible* oznacza, że od uruchomienia aplikacji dany obiekt nie był jeszcze widziany przez system śledzący; pole czerwone – obiekt był widziany, ale zniknął z pola widzenia; pole zielone – obiekt jest obecnie widziany i śledzony. Szare pole przycisku kontrolera Flystick oznacza, że dany przycisk nie jest obecnie wciśnięty; pole niebieskie – przycisk jest wciśnięty.



Rys. 1.7. *Input* – zakładka okna *LZWPlib* z podglądem danych od systemu śledzenia

Pakiet LZWPlib zawiera prefabrykat *TrackedObject* umożliwiający łatwe powiązanie obiektów sceny z zadanym obiektem śledzonym (z pozą). Zawiera on edytowalny skrypt o tej samej nazwie, w którym – w oknie inspektora – ustawić można za jaką pozą podążać ma ten obiekt. Wybrać można typ (ogólna poza, Flystick, okulary, dłoń) i indeks pozy z odpowiadającej mu listy (*Lzwp.input.poses*, *Lzwp.input.flysticks*, *Lzwp.input.glasses* lub *Lzwp.input.hands*). Domyślne ustawienie to pierwszy kontroler Flystick. Pozycja i orientacja obiektu, do którego podpięty jest ten skrypt, automatycznie aktualizowana będzie (we wszystkich instancjach aplikacji) na podstawie wartości ze wskazanej pozy. Nie należy obiektów takich umieszczać w innych obiektach niestatycznych – może to powodować drżenie lub przeskakiwanie obiektu.

W oknie inspektora można również podpiąć akcje wywoływane w momencie utraty widoczności śledzonej pozy (*OnTrackingLost*) oraz odzyskania jej – pojawienia się pozy w zasięgu śledzenia (*OnTrackingAcquired*). Domyślnie ustawione tam akcje powodują włączenie lub wyłączenie obiektu *HidingContainer*, będącego dzieckiem obiektu prefabrykatu.

Przykładowy kod demonstrujący odczyt pozy pierwszego kontrolera Flystick, stanu wciśnięcia jego spustu oraz wychylenia joysticka:

```
LzwpPose pose = new LzwpPose();
```

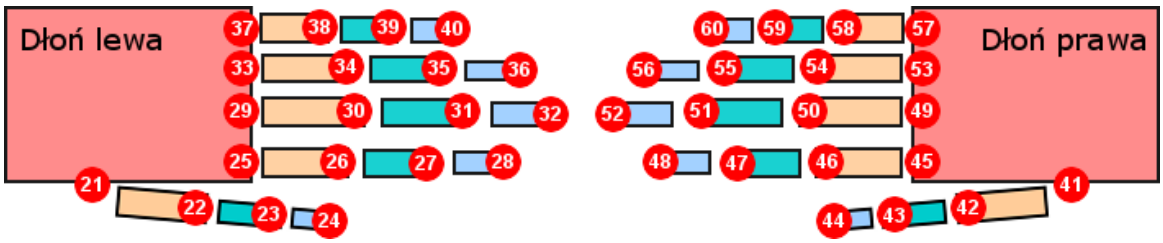
```
void Start()
{
    Lzwp.AddAfterInitializedAction(LzwpReady);
}

void LzwpReady()
{
    if (Lzwp.input.flysticks.Count > 0)
    {
        pose = Lzwp.input.flysticks[0].pose;
        pose.OnTrackingStateChanged += TrackingStateChanged;

        Lzwp.input.PosesUpdated += UpdatePose;

        Lzwp.input.flysticks[0].GetButton(LzwpInput.Flystick.ButtonID.Fire)
            .OnPress += () => { Lzwp.debug.Log("FIRE!"); };
    }
}

void UpdatePose(bool hasNewData)
{
    transform.position = pose.position;
    transform.rotation = pose.rotation;
}

void TrackingStateChanged(bool tracked)
{
    Lzwp.debug.Log("Tracked object is visible: {0}", tracked);
}

void Update()
{
    if (Lzwp.initialized)
    {
        if (Lzwp.input.flysticks.Count > 0 && Lzwp.input.flysticks[0]
            .GetButton(LzwpInput.Flystick.ButtonID.Fire).isActive)
        {
            // user holds down the trigger (Fire button)
        }
    }
}
```

LZWPlib pozwala także wykorzystać dane z aplikacji ART-Human [8], która korzystając z póz śledzonych targetów z zestawu śledzenia ciała oraz z algorytmów kinematyki odwrotnej (ang. *inverse kinematics, IK*) wyznacza w czasie rzeczywistym pozę ciała (szkielet), czyli pozy poszczególnych kości. Każdy nowy użytkownik musi na początku utworzyć swój profil (model) i przejść proces kalibracji, podczas którego wyliczane są długości poszczególnych kości (z dokładnością do 1 cm). Na dłonie może założyć zwyczajne targety (np. HT23 i HT25) lub zestaw śledzenia dłoni i palców, co rozszerzy wynikowe dane o pozy paliczków. Do danych tych można również dołączyć targety niewchodzące w skład zestawu śledzenia ciała, określane mianem „narzędzi”, np. kontrolery Flystick.

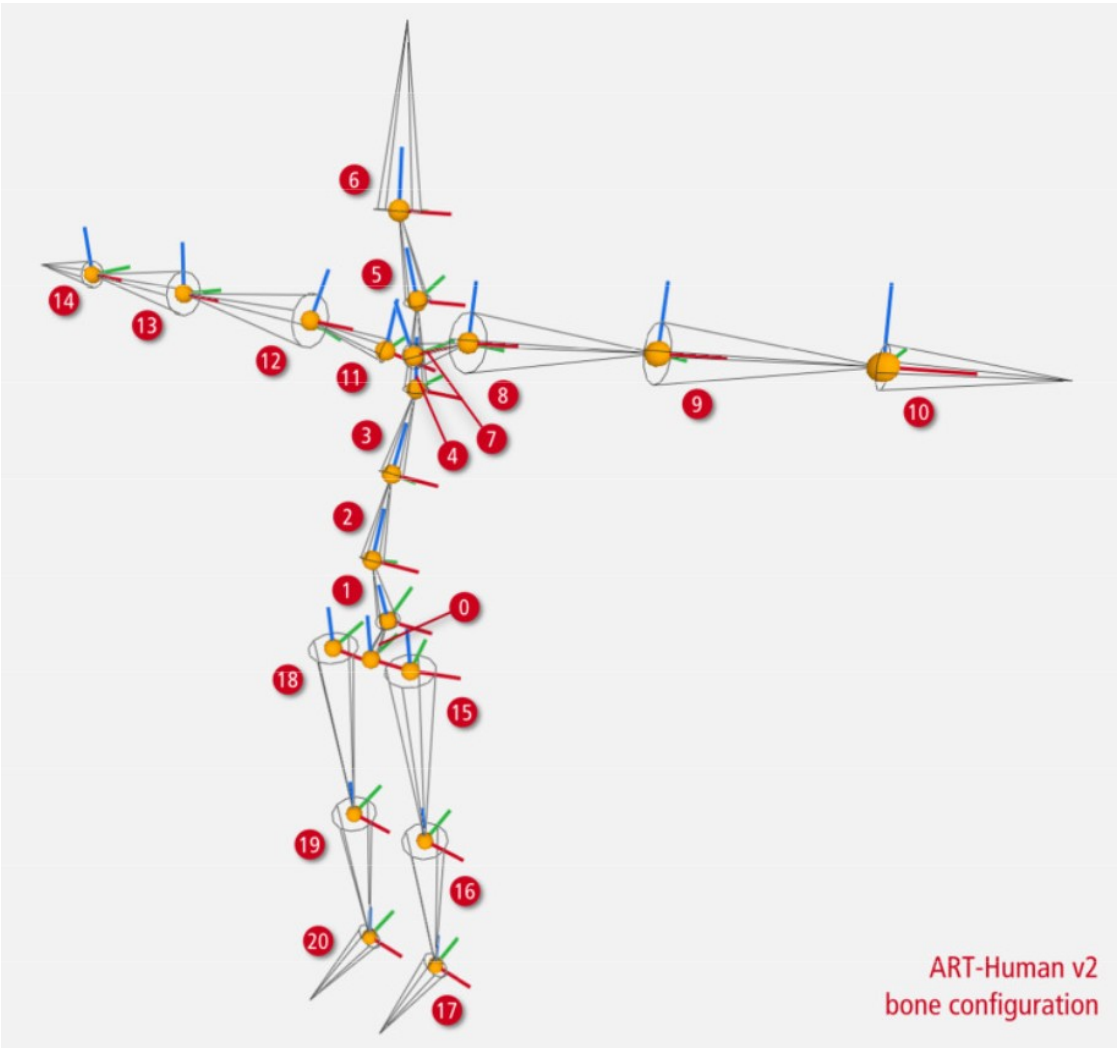
ART-Human to oddzielnie uruchamiany program, który oczekuje danych z systemu DTrack (nasłuchuje na zadanym porcie, domyślnie 7000), a wyliczone dane udostępnia w wybranym

standardzie (domyślny i zalecany standard to ART-Human v2) na jeden z kilku dostępnych sposobów, np. wysyłając ramki poprzez protokół UDP na zdefiniowany adres i port (domyślnie adres lokalny i port 6000) w formacie takim jak DTrack.

Przyporządkowanie identyfikatorów do poszczególnych kości wg standardu ART-Human v2 przedstawione zostało na rys. 1.8 i 1.9 oraz w tabeli 1.1. Pozy kości udostępniane przez LZWPlib przekształcone są do lewoskrętnego układu współrzędnych.



Rys. 1.8. Przyporządkowanie identyfikatorów do kości dłoni w standardzie ART-Human v2



Rys. 1.9. Przyporządkowanie identyfikatorów do kości szkieletowych w standardzie ART-Human v2 [8]

Tabela 1.1. Przyporządkowanie identyfikatorów do kości szkieletowych i kości dłoni w standardzie ART-Human v2

ID kości	Nazwa kości
0	miednica
1	dolny odcinek kręgosłupa lędźwiowego
2	górny odcinek kręgosłupa lędźwiowego
3	dolny odcinek kręgosłupa piersiowego
4	górny odcinek kręgosłupa piersiowego
5	szyja
6	czaszka (głowa)
7	lewe ramię (obojczyk)
8	kość ramienna lewej ręki
9	kość łokciowa/promieniowa lewej ręki
10	lewa dłoń
11	prawe ramię (obojczyk)
12	kość ramienna prawej ręki
13	kość łokciowa/promieniowa prawej ręki
14	prawa dłoń
15	kość udowa lewej nogi
16	kość piszczelowa/strzałkowa lewej nogi
17	lewa stopa
18	kość udowa prawej nogi
19	kość piszczelowa/strzałkowa prawej nogi
20	prawa stopa
21 – 24	kciuk lewej dłoni – kość nadgarstka, śródręcza, paliczek bliższy i dalszy
25 – 28	palec wskazujący lewej dłoni – kość śródręcza, paliczki: bliższy, środkowy, dalszy
29 – 32	palec środkowy lewej dłoni – kość śródręcza, paliczki: bliższy, środkowy, dalszy
33 – 36	palec serdeczny lewej dłoni – kość śródręcza, paliczki: bliższy, środkowy, dalszy
37 – 40	palec mały lewej dłoni – kość śródręcza, paliczki: bliższy, środkowy, dalszy

41 – 44	kciuk prawej dłoni – kość nadgarstka, śródręcza, paliczek bliższy i dalszy
45 – 48	palec wskazujący prawej dłoni – kość śródręcza, paliczki: bliższy, środkowy, dalszy
49 – 52	palec środkowy prawej dłoni – kość śródręcza, paliczki: bliższy, środkowy, dalszy
53 – 56	palec serdeczny prawej dłoni – kość śródręcza, paliczki: bliższy, środkowy, dalszy
57 – 60	palec mały prawej dłoni – kość śródręcza, paliczki: bliższy, środkowy, dalszy

Przy inicjalizacji biblioteki LZWPlib dla każdego połączenia z aplikacją ART-Human, zdefiniowanego w konfiguracji (na liście *input.arthumans*), tworzony jest obiekt *ArtHuman*, który dodawany jest do tablicy *Lzwp.input.arthumans*. Następnie, jeśli w konfiguracji obsługa danego połączenia jest odblokowana (parametr *Enabled*), rozpoczyna się asynchroniczne odbieranie i przetwarzanie danych od tej aplikacji. LZWPlib nasłuchuje na zdefiniowanym w konfiguracji porcie UDP (parametr *Data Port*, domyślnie: 6000). Konfiguracja zawiera też parametr pozwalający określić położenie początku układu współrzędnych aplikacji ART-Human względem początku układu współrzędnych jaskini – *Offset From Origin*.

Po odebraniu ramki obiektami *Human* uzupełniana jest lista *Lzwp.input.humans* (w MidiCAVE'ie, ze względu na ograniczenia licencyjne oraz na dostępność tylko jednego zestawu śledzenia ciała, będzie to maksymalnie jeden obiekt). Jeśli obiekt był już na tej liście – jego dane zostają zaktualizowane. Obecnie obiekt *Human* zawiera jedynie pole *joints* – 200-elementową tablicę (taki limit określony został w SDK producenta) obiektów typu *LzwpPose*, określających pozy kości o identyfikatorach odpowiadających indeksom tej tablicy.

Plik konfiguracyjny pozwala zdefiniować płaszczyzny przeszkód fizycznych (parametr *Obstacle Walls*), na których będzie wyświetlane ostrzeżenie o zbliżaniu się do nich śledzonych obiektów (rys. 1.10). Zwykle są tymi płaszczyznami po prostu ściany jaskini (czasami łączące w sobie kilka ekranów, jak np. w BigCAVE'ie). Można jednak pomyśleć o sytuacji, w której do jaskini wstawiono jakiś obiekt fizyczny i wokół niego również chcemy „rozstawić" takie ostrzeżenia. LZWPlib zapewnia wyświetlanie ostrzeżeń dla wszystkich śledzonych obiektów (okularów, kontrolerów Flystick, innych targetów). Parametr konfiguracyjny *Obstacle Wall Warning Scale* określa skalę wyświetlanych ostrzeżeń.

Rys. 1.10. Ostrzeżenie o zbliżaniu się okularów do ściany jaskini

1.13. *Synchronizacja*

Między innymi ze względu na prostotę użycia, kompatybilność z rozwiązaniem wykorzystywanym wcześniej (przydatną przy aktualizacji starych aplikacji) oraz wydzielenie implementacji nowych mechanizmów sieciowych do innego projektu dyplomowego, w pakiecie LZWPlib zaimplementowane zostało wsparcie synchronizacji stanu aplikacji wykorzystujące stare mechanizmy sieciowe silnika Unity [9]. Obecnie są one już przestarzałe i zostały całkowicie usunięte w wersji 2018.2 edytora [10]. Trzeba jednak przyznać, że spełniały one swoją rolę i implementacja synchronizacji z ich użyciem była stosunkowo prosta.

Po uruchomieniu aplikacji podejmowana jest próba zestawienia połączenia sieciowego pomiędzy instancjami działającymi jako klienty, a autorytarnym serwerem. Tryb działania aplikacji określany jest poprzez porównanie ustalonego na wcześniejszym etapie inicjalizacji biblioteki identyfikatora bieżącej instancji (*Lzwp.instanceID*) z identyfikatorem instancji głównej, zdefiniowanym przez parametr konfiguracyjny *sync.masterInstanceID* (domyślnie: 0). Tryb ten sprawdzić można za pomocą właściwości *isMaster*, która określa, czy aplikacja uruchomiona została w trybie serwera:

```
bool inServerMode = Lzwp.sync.isMaster;
```

Aplikacja uruchamiana jako serwer nasłuchuje na porcie zdefiniowanym przez parametr konfiguracyjny *sync.port* (domyślnie: 54321) i akceptuje maksymalną liczbę połączeń, jaką określa parametr *sync.maxConnections* (domyślnie: 32). Parametr *sync.sendRate* definiuje częstotliwość aktualizacji stanu aplikacji (domyślnie: 60 Hz). Aplikacje klienckie podejmują próby połączenia się z serwerem korzystając z adresu zdefiniowanego w parametrze *sync.serverIP* (domyślnie: 127.0.0.1) oraz z portu określonego przez wspomniany już parametr *sync.port*. Maksymalna liczba prób połączenia, jaką wykonuje klient, określona jest parametrem *sync.maxConnectionTries* (domyślnie: 5 prób). Po każdej nieudanej próbie następuje jednosekundowa przerwa. Jeśli po przekroczeniu limitu prób nie uda się nawiązać połączenia, następuje zamknięcie programu lub powrót do trybu edycji, jeśli aplikacja uruchomiona była w edytorze. Podobnie dzieje się w przypadku zerwania połączenia z serwerem, jeśli parametr konfiguracyjny *sync.quitOnDisconnection* ma ustawioną wartość *true* (domyślną). Mechanizm ten wykorzystywany jest do zamykania wszystkich instancji danej aplikacji działających na klastrze, co inicjowane jest wyłączeniem aplikacji działającej w trybie serwera (np. poprzez skrót klawiszowy Alt + F4).

LZWPlib daje możliwość swobodnego przełączania scen aplikacji. Wydzielone zostały dwie grupy sieciowe:

- grupa 1 – obsługująca komunikację dotyczącą ładowania scen;
- grupa 0 (domyślna) – przeznaczona do obsługi pozostałego ruchu sieciowego.

Ładowanie nowej sceny (lub przeładowanie obecnej) może być zainicjowane jedynie przez aplikację działającą w trybie serwera. Moduł synchronizacji udostępnia twórcy aplikacji następujące metody związane z ładowaniem scen:

- *GetActiveScene()* – zwraca aktualnie załadowaną scenę (obiekt typu *UnityEngine.SceneManagement.Scene*);
- *LoadScene(string sceneName)* – ładuje na wszystkich instancjach aplikacji scenę o nazwie przekazanej jako parametr tej metody; ładowanie działa jedynie wtedy, gdy zostanie wywołane z instancji serwerowej;
- *ReloadScene()* – przeładowuje bieżącą scenę; działa jedynie wtedy, gdy zostanie wywołane z instancji serwerowej.

Udostępniona jest również akcja *OnSceneLoaded*, która wywoływana jest po zakończeniu ładowania nowej sceny bądź też przeładowania bieżącej.

LZWPlib synchronizuje automatycznie:

- aktualną pozycję, rotację i widoczność wszystkich póz (obiektów *LzwpPose*) znajdujących się na liście *Lzwp.input.poses* – dzięki temu możliwe jest m.in. odpowiednie dopasowanie perspektywy oraz wyświetlanie ostrzeżeń o zbliżaniu się do przeszkód fizycznych, korzysta z tego także skrypt *TrackedObject* używany m.in. przez prefabrykat o tej samej nazwie;
- pozycję i rotację reprezentacji jaskini w świecie wirtualnym (poprzez komponent *NetworkView* przypięty do obiektu ze skryptem *LzwpOrigin*);
- czas mierzony w sekundach i liczony od momentu uruchomienia się aplikacji serwerowej.

Synchronizację pozostałych elementów – co najmniej tych wpływających na zmiany wizualne – twórca aplikacji musi zaimplementować samodzielnie. Może w tym celu skorzystać z mechanizmów zapewnianych przez komponent *NetworkView* silnika Unity. Podpięcie tego komponentu do obiektu synchronizowało jego pozycję, rotację i skalę, gdyż jako element obserwowany (pole *Observed*) domyślnie ustawiany był komponent *Transform*. Obserwowana mogła być również animacja (komponent *Animation*), co synchronizowało czas, wagi, prędkości i inne właściwości jej stanów, a także ciała sztywne (komponent *Rigidbody*), w których synchronizowana była pozycja, rotacja, prędkość oraz prędkość kątowa. Podpięcie obserwowania skryptu dziedziczącego po klasie *MonoBehaviour* powodowało wywoływanie w nim metody *OnSerializeNetworkView*, za pomocą której można było pisać bezpośrednio do strumienia sieciowego po stronie serwera, a po stronie klienckiej z tego strumienia czytać. Obiekty *NetworkView* posiadały również metodę *RPC*, służącą – jak sama nazwa wskazuje – do zdalnego wywoływania procedur (ang. *remote procedure call*). Pozwalało to wywołać na wskazanych węzłach (np. na wszystkich klientach) dowolną metodę oznaczoną atrybutem *RPC* i przekazać do niej argumenty typu *int*, *float*, *string*, *NetworkPlayer*, *NetworkViewID*, *Vector3* lub *Quaternion* [11]. Przydatne mogło być również wykorzystanie statycznej metody *Network.Instantiate* do tworzenia nowych obiektów podczas działania aplikacji. Wywołanie tej metody na serwerze powodowało utworzenie obiektu we wszystkich podłączonych instancjach klienckich. Analogicznie – do usunięcia obiektu na wszystkich instancjach można było użyć metody *Network.Destroy*.

Pamiętać jednak należy, że cała ta synchronizacja powinna odbywać się w ramach grupy sieciowej oznaczonej numerem 0, aby poprawnie działał zaimplementowany w LZWPlibie mechanizm ładowania scen.

Synchronizowany przez bibliotekę czas, liczony od momentu uruchomienia się aplikacji serwerowej, można odczytać za pośrednictwem pola *time*:

```
float timeSinceStartup = Lzwp.sync.time;
```

Dostępny jest on również w programach cieniujących po zadeklarowaniu odpowiedniej zmiennej:

```
uniform float4 _LzwpTime;
uniform float4 _LzwpSinTime;
uniform float4 _LzwpCosTime;
```

Deklaracje te można również załączyć korzystając z pliku *Assets\LZWPlib\Shaders\LZWPlibCG.cginc*:

```
#include "Assets/LZWPlib/Shaders/LZWPlibCG.cginc"
```

_LzwpTime, *_LzwpSinTime* i *_LzwpCosTime* odpowiadają zmiennym *_Time*, *_SinTime* i *_CosTime* standardowo dostępnym w Unity poprzez załączenie pliku *UnityCG.cginc*, w którym z kolei załączony jest plik *UnityShaderVariables.cginc*, w którym zmienne te są zadeklarowane [12]. Każda z nich jest 4-elementowym wektorem, którego kolejne składowe zawierają wartość zsynchronizowanego czasu przemnożoną przez odpowiedni współczynnik, a w przypadku dwóch ostatnich zmiennych – przepuszczoną dodatkowo przez odpowiednią funkcję trygonometryczną. Przyjmując t jako wartość zsynchronizowanego czasu, składowe wektorów przedstawić można następująco:

$$LzwpTime = \left[\frac{t}{20} \quad t \quad 2t \quad 3t\right],$$

$$LzwpSinTime = \left[\sin\frac{t}{8} \quad \sin\frac{t}{4} \quad \sin\frac{t}{2} \quad \sin t\right],$$

$$LzwpCosTime = \left[\cos\frac{t}{8} \quad \cos\frac{t}{4} \quad \cos\frac{t}{2} \quad \cos t\right].$$

1.14. *Przygotowanie projektu*

Przygotowanie nowego lub dostosowanie istniejącego projektu do pracy w środowisku jaskiń rzeczywistości wirtualnej z użyciem pakietu LZWPlib może przebiegać w następujących krokach:

- import pakietu LZWPlib;
- dostosowanie wymaganych ustawień projektu poprzez kliknięcie przycisku *Adjust all settings* w zakładce *Project settings* okna *LZWPlib*;
- dostosowanie scen projektu:
  - przy tworzeniu nowej sceny posłużyć się można szablonem *SceneTemplate\SceneTemplate.unity*;

- w przypadku adaptacji sceny już istniejącej:
  - dodanie prefabrykatu *[LZWPlib]*;
  - dodanie prefabrykatu reprezentującego jaskinię – *[LzwpOrigin]* lub *[LzwpOrigin_M]* – i odpowiednie ustawienie go na scenie; w przypadku użycia drugiego z prefabrykatów – dostosowanie ustawień poruszania się;
  - wykorzystanie prefabrykatu z kamerami – *[CameraContainer]* – lub odpowiednie ustawienie istniejących kamer i dodanie do nich skryptu LzwpCamera;
  - usunięcie nadmiarowych obiektów, np. typu *AudioListener*;
- zaimplementowanie sterowania z użyciem kontrolerów Flystick (można wykorzystać prefabrykat *TrackedObject*) lub sferycznego symulatora chodu;
- dodanie synchronizacji stanu aplikacji – co najmniej elementów wpływających na zmiany wizualne;
- wybudowanie aplikacji.

Ogólne zalecenia odnośnie tworzenia aplikacji VR:

- właściwe stosowanie skali – 1 jednostka na scenie odpowiada 1 metrowi w rzeczywistości;
- włączenie prostego wygładzania krawędzi (ang. *Anti-Aliasing, AA*) jako efekt przetwarzania końcowego (ang. *postprocessing*);
- niestosowanie lub mocno ograniczone użycie efektów przetwarzania końcowego ciężkich obliczeniowo, takich jak np. okluzja otoczenia (ang. *Ambient Occlusion, AO*) czy odbicia w przestrzeni ekranu (ang. *Screen Space Reflections, SSR*);
- niestosowanie efektów przetwarzania końcowego powodujących niespójność obrazu na łączeniach ekranów, takich jak np. winietowanie (ang. *Vignette*);
- niestosowanie efektów symulujących optykę kamery, takich jak aberracja chromatyczna (ang. *chromatic aberration*), rozmycie ruchu (ang. m*otion blur*), głębia ostrości (ang. *Depth of Field, DoF*) – niektóre z nich mogą powodować złe samopoczucie użytkownika;
- niestosowanie efektów przetwarzania końcowego, które można uzyskać naturalnie, takich jak adaptacja oka do natężenia światła (ang. *eye adaptation*);
- niestosowanie trudnych do zsynchronizowania efektów cząsteczkowych;
- implementacja poruszania się skokowego lub ruchem jednostajnym.

1.15. *Namiastka pakietu*

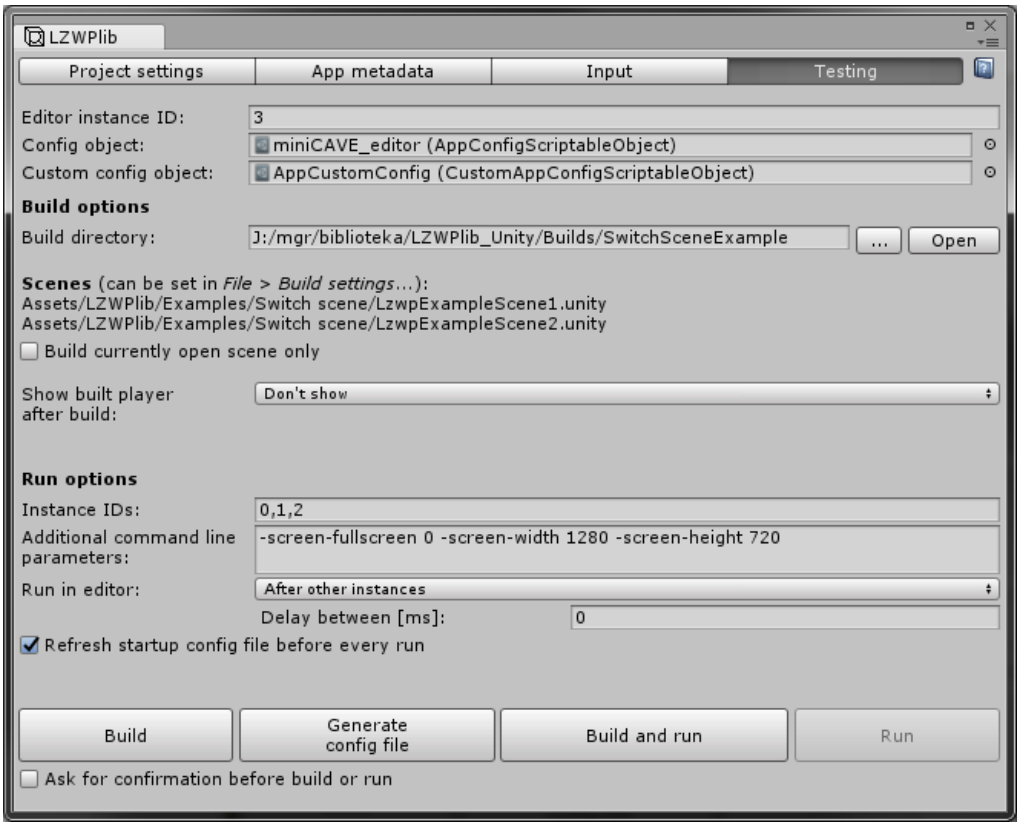
Podjęto decyzję o nierozpowszechnianiu kompletnego pakietu poza LZWP. Przygotowano więc okrojoną wersję LZWPliba, w której wycięte zostały funkcje zazwyczaj i tak nieprzydatne w standardowym środowisku komputerowym. W wersji mock pakietu brakuje obsługi śledzenia (pozy pozostają nieruchome w pozycji startowej) oraz dopasowania renderowanego obrazu do geometrii ekranów. Aby przejść z wersji okrojonej na pełną, wystarczy jedynie podmienić plik biblioteki: *LZWPlib.dll*. Podmienić go można już w katalogu zasobów projektu (*Assets\LZWPlib\Plugins\LZWPlib.dll*), co może być przydatne podczas testowania aplikacji w laboratorium, bądź też dopiero po zbudowaniu aplikacji (będzie się znajdował w katalogu *[NazwaAplikacji]_Data\Managed*).

Korzystając z okienka informacyjnego, dostępnego w menu głównym w pozycji *LZWPlib – About...*, sprawdzić można, czy w edytorze używana jest wersja pełna, czy okrojona (ta druga zawiera napis „mock”). Informacja o wersji wypisywana jest również do logu.

1.16. *Testowanie na wielu instancjach*

Aplikacje przygotowane przy użyciu LZWPliba mogą zostać uruchomione w kilku instancjach na pojedynczej maszynie. Dzięki temu możliwe jest stosunkowo wygodne testowanie synchronizacji stanu aplikacji pomiędzy instancją serwerową, a instancjami klienckimi. Proces testowania ułatwić ma zakładka *Testing* okna *LZWPlib* (rys. 1.11).

W górnej jej części wskazać można obiekt konfiguracji podstawowej (*Config object*) oraz dodatkowej (*Custom config object*), które używane będą podczas testów, oraz identyfikator instancji (*Editor instance ID*), którego używać będzie aplikacja uruchamiana w edytorze. Pamiętać należy, że na klastrze uruchamiane są identyczne aplikacje z takim samym plikiem konfiguracyjnym, a wczytanie i wykorzystanie odpowiedniej konfiguracji uzależnione jest od identyfikatora instancji.



Rys. 1.11. *Testing* – zakładka okna *LZWPlib* ułatwiająca testowanie aplikacji na wielu instancjach

Domyślnie w edytorze uruchamiana jest instancja oznaczona numerem 0, co standardowo odpowiada instancji serwerowej (którą określa parametr *Sync – Master Instance ID* w wybranej konfiguracji podstawowej). Wskazanie innego numeru umożliwia podłączenie edytora jako instancji klienckiej do uruchomionej w oddzielnym procesie (lub nawet na oddzielnej maszynie) aplikacji serwerowej.

Po dostosowaniu pozostałych ustawień na tej zakładce, wykorzystywać ją można do szybkiego budowania i uruchamiania wielu instancji aplikacji na bieżącej maszynie. W polu *Build directory* wskazać można katalog, w którym zapisana zostanie zbudowana aplikacja. W celu jego wybrania posłużyć się można przyciskiem po prawej stronie tego pola. Przycisk *Open* otwiera wskazany katalog. Jeśli zaznaczone zostanie pole *Build currently open scene only*, to zbudowana aplikacja będzie zawierała tylko jedną scenę – tę, która wczytana była w edytorze w momencie rozpoczęcia procesu budowania. W przeciwnym wypadku użyte zostaną sceny znajdujące się na liście w oknie ustawień budowania (menu *File – Build Settings...*). Z listy *Show built player after build* wybrać można, czy po zbudowaniu aplikacji ma zostać otwarty katalog ją zawierający. Budowanie aplikacji rozpocząć można przyciskiem *Build*.

W sekcji *Run options* dostosować można opcje automatycznego uruchamiania wielu instancji. W polu *Instance IDs* podać należy oddzielone przecinkami identyfikatory dla instancji, które uruchomione zostaną w oddzielnych procesach. Identyfikatory te przekazane będą jako argumenty uruchomieniowe procesów. W polu *Additional command line parameters* wskazać można dodatkowe argumenty, np. wyłączające tryb pełnoekranowy („-screen-fullscreen 0”) czy też ustawiające rozmiary okna („-screen-width 1280 -screen-height 720”). W przypadku częstego uruchamiania wielu instancji można dodatkowo ustawić pozycję okna każdej z nich w konfiguracji podstawowej, korzystając z parametrów *Display – Set Window Pos* oraz *Display – Window Pos*. Aby ułatwić identyfikację okien, do tekstu w ich belkach tytułowych po inicjalizacji biblioteki dopisany zostaje numer instancji (w nawiasie kwadratowym).

Pole *Run in editor* pozwala wybrać, czy poza instancjami wskazanymi na liście uruchamiana ma być również aplikacja w edytorze, a jeśli tak – czy ma się to stać przed, czy po uruchomieniu instancji z listy. W polu *Delay between* wskazać można opcjonalny odstęp w milisekundach pomiędzy uruchomieniem instancji w edytorze i pierwszej lub ostatniej instancji z listy.

Podczas wybierania identyfikatorów instancji do uruchomienia (tych na liście *Instances IDs* oraz w polu *Editor instance ID*) pamiętać należy, że tylko jedna z nich może być uruchomiona w trybie serwera.

Przycisk *Build and run* powoduje zbudowanie aplikacji, wygenerowanie w jej katalogu startowego pliku konfiguracyjnego oraz uruchomienie wskazanych instancji. Jeśli aplikacja została już zbudowana, samo uruchomienie (opcjonalnie poprzedzone regeneracją pliku konfiguracyjnego, jeśli zaznaczono opcję *Refresh startup config file before every run*) wielu instancji wywołać można przyciskiem *Run*. Przycisk *Generate config file* pozwala wygenerować na nowo sam plik konfiguracyjny. Aby uniknąć przypadkowego uruchomienia lub rozpoczęcia budowania aplikacji, zaznaczyć można opcję *Ask for confirmation before build or run*, co skutkowało będzie wyświetleniem prośby o potwierdzenie chęci zbudowania lub uruchomienia aplikacji po kliknięciu przycisku *Build*, *Build and run* lub *Run*.

1.17. *Sceny przykładowe*

Pakiet LZWPlib zawiera prosty szablon sceny, który może zostać wykorzystany przy tworzeniu nowego projektu lub dodawaniu nowej sceny do projektu już istniejącego. Szablon zawarty jest w pliku *Assets\LZWPlib\SceneTemplate\SceneTemplate.unity*. Scena ta zawiera podstawowe komponenty:

- *[LZWPlib]* – główny obiekt inicjalizujący bibliotekę (wymagany);
- *[LzwpOrigin_M]* – reprezentacja jaskini w świecie wirtualnym z możliwością przemieszczania się – chodzenia i latania; jeden z dwóch dostępnych prefabrykatów zawierających wymaganą reprezentację jaskini;
- *[CameraContainer]* – kontener z dwoma obiektami kamer („wzorem kamer") – dla lewego i prawego oka; „wzór kamer" jest wymagany;
- *Flystick1* – obiekt podążający za pierwszym śledzonym kontrolerem;
- *Flystick2* – obiekt podążający za drugim śledzonym kontrolerem;
- *Directional Light* – domyślne światło sceny;
- *Ground* – podłoże (kwadrat o boku długości 10 m), po którym chodzić może wirtualna reprezentacja użytkownika.

LZWPlib zawiera również katalog *Assets\LZWPlib\Examples* z przykładami demonstrującymi implementację użycia dostępnych funkcji pakietu. Korzystając z nich należy mieć na uwadze, że nie są to kompletne sceny, które mogłyby w pełni działać w środowisku typu CAVE, a jedynie stosunkowo minimalistyczne prezentacje poszczególnych funkcji – w szczególności nie uwzględniające synchronizacji stanu pomiędzy węzłami jaskini. Każdy przykład zawarty jest w osobnym katalogu, z których każdy zawiera edytowalne skrypty, w których można podejrzeć implementację wykorzystania poszczególnych funkcji, oraz co najmniej jedną scenę z przykładem. Każda scena ma w hierarchii obiektów pozycję *[README]*. Jej zaznaczenie wyświetla w oknie inspektora panel ze wstępnymi informacjami dotyczącymi danej sceny. Niektóre przykłady mogą wymagać odpowiedniego przygotowania projektu w celu ich zademonstrowania. Wówczas wspomniany panel zawierać będzie instrukcję przygotowania przykładu; może również zawierać przyciski *Prepare example* oraz *Clear example*, odpowiadające odpowiednio za automatyczne jego przygotowanie oraz za wycofanie zmian przygotowujących.

1.17.1. *Pisanie do logu i konsoli*

Scena *Logging\Logging_LzwpExample* zawiera skrypt *LoggingExample* przypięty do obiektu *Logging*, demonstrujący użycie funkcji modułu *Lzwp.debug* pozwalających pisać do logu oraz okna konsoli. Zawarty w nim kod:

```
Lzwp.debug.Log("Some string");
Lzwp.debug.Log("Log with <b>rich</b> text.");
Lzwp.debug.Log("PI = {0}  TAU = {1}", Mathf.PI, 2f * Mathf.PI);
Lzwp.debug.Log(Vector3.right);
Lzwp.debug.LogWithContext(this, "Log with context");

Lzwp.debug.Warning("Some warning string");
Lzwp.debug.Warning("Unknown type: {0}", typeof(LoggingExample));
```

```
Lzwp.debug.Error("Some error!");

try
{
    throw new Exception("Some exception");
}
catch (Exception ex)
{
    Lzwp.debug.Exception(ex);
    Lzwp.debug.ExceptionWithContext(this, ex);
}
```

po uruchomieniu aplikacji spowoduje pojawienie się następujących wpisów (oczywiście czasy mogą być inne):

```
<< wpisy pochodzące z inicjalizacji biblioteki >>
[16:31:02.705] Some string
[16:31:02.705] Log with rich text.
[16:31:02.706] PI = 3.141593  TAU = 6.283185
[16:31:02.706] (1.0, 0.0, 0.0)
[16:31:02.707] Log with context
[16:31:02.707] Some warning string
[16:31:02.708] Unknown type: LoggingExample
[16:31:02.708] Some error!
[16:31:02.709] EXCEPTION:
Exception: Some exception
[16:31:02.709] EXCEPTION:
Exception: Some exception
```

1.17.2. *Konfiguracja*

Przykład Custom Config*CustomConfig_LzwpExample* wymaga przygotowania – wygenerowania dodatkowego pliku skryptu *CustomConfigUsage_LzwpExample.cs*, w którym zawarta będzie przykładowa konfiguracja dodatkowa. Do jego wygenerowania użyć należy przycisku *Prepare example*, do usunięcia — *Clear example*. Wygenerowany plik zawiera rozszerzenie klasy cząstkowej *CustomAppConfig*:

```
public partial class CustomAppConfig
{
    [Serializable]
    public class CubeRotation
    {
        public bool rotate = true;
        public float speed = 20f;

        [JsonConverter(typeof(LzwpUtils.Converters.Vector3Converter))]
        public Vector3 axis = Vector3.one;
    }
```

```
    public CubeRotation cubeRotation = new CubeRotation();

    public int someInt = 1337;
    public string someString = "LZWP!";
}
```

Do konfiguracji dodawane zostaje pole *someString* typu *string*, pole *someInt* typu *int* oraz pole *cubeRotation*, opakowujące klasą *CubeRotation* konfigurację dotyczącą pewnego sześcianu. Istotne jest opatrzenie tej klasy atrybutem *Serializable*. Klasa zawiera pola typu *bool* i *float*, a także pole typu *Vector3*, które wymaga opatrzenia atrybutem określającym odpowiedni dla niego konwerter – w tym przypadku *Vector3Converter*. Tak zdefiniowane pola konfiguracji będą mógły być edytowane w oknach edytora, pojawią się również w pliku konfiguracyjnym aplikacji (*LzwpStartupConfig.json*).

Klasa *CustomConfig_LzwpExample* zawiera przykład wykorzystania zdefiniowanych wcześniej dodatkowych pól. Jest ona podzielona na dwie części – druga z nich znajduje się w pliku *CustomConfig_LzwpExample.cs*, który przypięty jest do obiektu *GameManager* na przykładowej scenie i w którym zdefiniowane jest pole *cube* typu *Transform*, do którego w inspektorze podpięty został znajdujący się na scenie sześcian (obiekt *Cube*). W metodzie *Start* tejże klasy do logu/konsoli wypisane zostają wartości pobrane z pól *someString*, *someInt* oraz z pola należącego do konfiguracji poruszania się, zdefiniowanej w pliku *Assets\LZWPlib\Scripts\MovementController.cs*:

```
void Start()
{
    Debug.Log("String from custom config: " +
      Lzwp.config.GetCustom().someString);
    Debug.Log("Int from custom config: " + Lzwp.config.GetCustom().someInt);

    // declared in: Assets/LZWPlib/Scripts/MovementController.cs
    Debug.Log("Movement type from custom config: " +
      Lzwp.config.GetCustom().movement.movementSettings.general.movementType);
}
```

W metodzie *Update* natomiast następuje obracanie podpiętego sześcianu z zadaną w konfiguracji prędkością i wokół zdefiniowanej w niej osi – o ile obracanie to jest w konfiguracji włączone:

```
void Update()
{
    var cfg = Lzwp.config.GetCustom().cubeRotation;

    if (cfg.rotate)
        cube.Rotate(cfg.axis.normalized * cfg.speed * Time.deltaTime);
}
```

Podczas działania aplikacji w edytorze wczytaną konfigurację, zdefiniowaną m.in. w tym przykładzie, zmieniać można korzystając z obiektu *[EditConfigInPlayMode]* znajdującego się na scenie.

1.17.3. *Śledzenie*

Prosty przykład *Tracking\Tracking_LzwpExample* ukazuje jak można podpiąć obiekty sceny do obiektów podążających za śledzonymi kontrolerami Flystick czy też okularami. Obiekt *Glasses* przypięty ma skrypt *TrackedObject*, w którym jako typ śledzonego obiektu ustawione są okulary, a jako indeks podano wartość 0 – obiekt ten będzie więc podążał za pierwszymi śledzonymi okularami (pierwszymi na liście targetów zapisanej w ustawieniach systemu śledzącego). Wewnątrz obiektu *Glasses* umieszczono celownik – obiekt *Crosshair* – przesunięty do przodu o 2 m. Będzie on więc wyświetlany zawsze na wprost okularów w takiej właśnie odległości. Na scenie znajdują się również dwa obiekty odpowiadające kontrolerom – *Flystick1* (z indeksem 0) i *Flystick2* (z indeksem 1). Jeden z nich będzie podążał za pierwszym śledzonym kontrolerem, a drugi – za drugim. Wewnątrz każdego z nich znajduje się obiekt *HidingContainer*, który w skrypcie *TargetObject* podpięty jest do akcji *OnTrackingAcquired* oraz *OnTrackingLost*, które wywołują na nim metodę *SetActive* odpowiednio z parametrem *true* lub *false*, co powoduje ukrycie tego obiektu (i obiektów znajdujących się wewnątrz tego kontenera, czyli w tym przypadku wskaźnika – obiekt *Pointer*) w momencie, gdy śledzony Flystick zniknie z pola widzenia systemu śledzącego, oraz ponowne pokazanie obiektu w momencie wejścia kontrolera w to pole. Zaznaczenie obiektu *[LzwpOrigin_M]* spowoduje podczas działania aplikacji w edytorze rysowanie w oknie sceny śledzonych póz przemieszczających się w pobliżu graficznej reprezentacji wybranej jaskini – odpowiada za to pole *Poses - Draw mode* w komponencie *Lzwp Origin Gizmos* podpiętym do tego obiektu.

1.17.4. Obsługa *kontrolerów Flystick*

Przykładowa scena *Input\Input_LzwpExample* demonstruje odczyt danych dotyczących przycisków i joysticków należących do dwóch kontrolerów Flystick. Zawiera dwie żółte tablice z małymi czerwonymi walcami pośrodku każdej z nich. Pozycja walca reprezentuje wychylenie gałki joysticka – np. wychylenie jej w przód spowoduje przesunięcie walca do górnej krawędzi tablicy. Przemieszczający się walec rysuje na tablicy ślad. Pod tablicami prezentowane są wartości liczbowe wychyleń obu joysticków. Obsługa joysticków zawarta jest w skrypcie *FlystickJoystick_InputExample*, który podpięty jest do obiektów *JoystickBoard 1* i *JoystickBoard 2*. W każdym z nich na odpowiedni indeks kontrolera ustawione jest pole *Flystick Idx*. W skrypcie za odczyt wychylenia joysticka w dwóch osiach odpowiadają początkowe linijki metody *FixedUpdate*:

```
float x = Lzwp.input.flysticks[flystickIdx].joysticks[0];  // horizontal
float y = Lzwp.input.flysticks[flystickIdx].joysticks[1];  // vertical
```

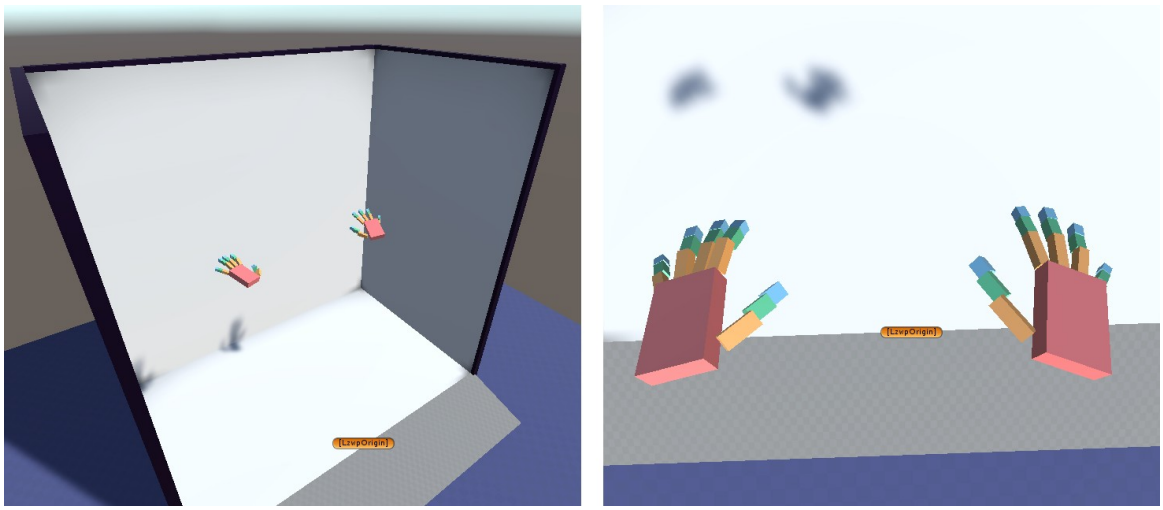
Scena zawiera również 12 białych kul - po 6 na każdy z kontrolerów Flystick. Odpowiadają one kolejnym przyciskom danego kontrolera: spust, przycisk nr 1, przycisk nr 2, przycisk nr 3, przycisk nr 4, przycisk w joysticku. Wciśnięcie danego przycisku zmienia kolor odpowiadającej mu kuli na zielony, puszczenie – na czerwony. Każda taka zmiana powoduje zwiększenie licznika zmian stanu, którego wartość prezentowana jest pod daną kulą. Przytrzymywany przycisk powoduje oscylację kuli w górę i w dół. Za obsługę przycisków odpowiada skrypt *MovingSphere_InputExample*, który przypięty

jest do każdej z kul. W polu *flystickIdx* znajduje się indeks danego kontrolera, natomiast pole *buttonID* określa przycisk obsługiwany przez daną instancję skryptu. W skrypcie tym istotne są następujące fragmenty kodu, przypisujące akcje do wciśnięcia i puszczenia przycisku oraz sprawdzające, czy w danej klatce wskazany przycisk jest wciśnięty:

```
void Start()
{
  // (...)

  Lzwp.input.flysticks[flystickIdx].GetButton(btnID).OnPress += OnPress;
  Lzwp.input.flysticks[flystickIdx].GetButton(btnID).OnRelease += OnRelease;
}


void Update()
{
  bool move = Lzwp.input.flysticks[flystickIdx].GetButton(buttonID).isActive;
  // (...)
}
```

1.17.5. *Śledzenie dłoni i palców*

Przykład *Fingertracking\Fingertracking_LzwpExample* pokazuje podpięcie systemu śledzenia dłoni i palców pod zestaw kolorowych prostopadłościanów, odpowiadający lewej i prawej dłoni oraz poszczególnym paliczkom (rys. 1.12). Poza wspomnianymi bryłami na scenie znajduje się też obiekt MidiCave, mający reprezentować średnią jaskinię LZWP, w której system ten jest dostępny. Do kontenera z dłońmi – *Hands* – podpięty jest skrypt *Hands_FingertrackingExample*, do którego z kolei podpięty jest obiekt lewej i prawej dłoni. Można w nim również wybrać, czy dłonie mają się przemieszczać, czy też ruchome mają być jedynie palce, a także czy ukryć obiekt dłoni, kiedy zniknie on z pola widzenia systemu śledzącego.



Rys. 1.12. Przykład podpięcia systemu śledzenia dłoni i palców

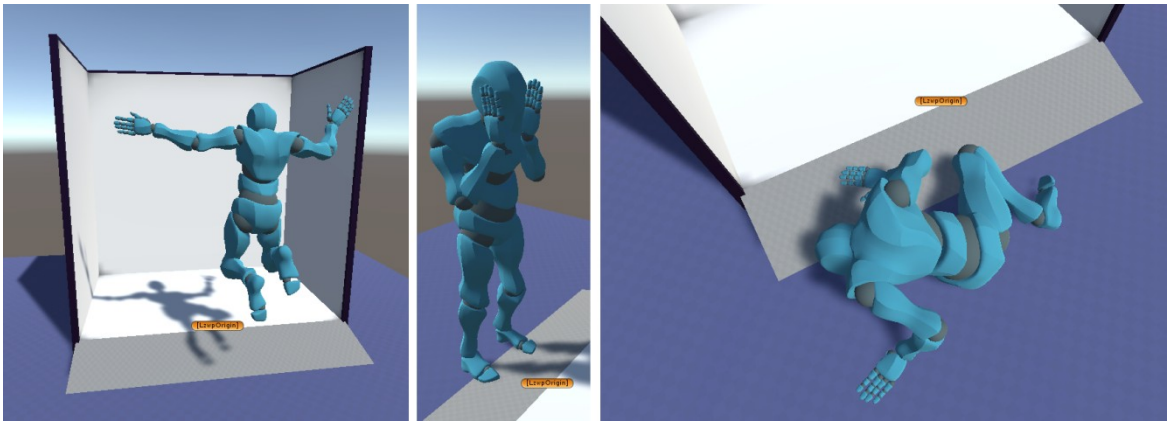
Aktualizacja modelu dłoni wykonywana jest w metodzie *Update*, której kod przedstawiono na poniższym listingu. Dla każdej śledzonej dłoni (linie nr 1 – 49) ustalany jest główny obiekt jej modelu (linie nr 3 – 5). Jeśli dłoń jest widziana przez system śledzenia (linia nr 7), to aktualizowana jest jej

pozycja i rotacja (linia nr 11 i 12) - o ile przemieszczanie dłoni jest włączone (linia nr 9), a następnie dla każdego palca (linie nr 15 – 44) pobierane są jego aktualne dane (linia nr 17) i ustawiane są jego 3 kolejne paliczki (linie nr 19 – 43). Dla każdego paliczka, za pomocą metody pomocniczej *GetFingerPhalanx*, pobierany jest odpowiadający mu obiekt sceny (linie nr 21 – 22), a następnie ustawiana jest jego skala (z wykorzystaniem danych o jego długości oraz o promieniu czubka palca; linie nr 24 – 28), lokalna pozycja (z wykorzystaniem danych o jego pozycji w układzie współrzędnych dłoni; linie nr 30 – 32) oraz lokalna rotacja (z wykorzystaniem danych o rotacji czubka palca i kątów pomiędzy kolejnymi paliczkami; linie nr 34 – 42). Jeśli włączone jest ukrywanie dłoni znajdujących się poza zasięgiem śledzenia, to dany obiekt dłoni jest włączany lub wyłączany, zależnie od jego aktualnej widoczności (linie nr 47 – 48).

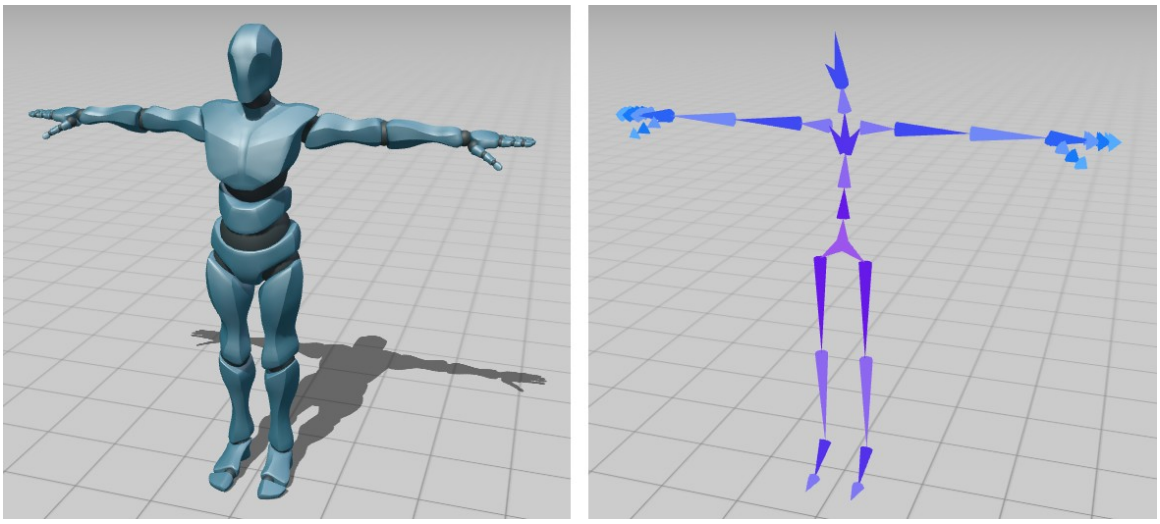
```
 1: for (int i = 0; i < Lzwp.input.hands.Count; i++)  // for each hand
 2: {
 3:   Transform tr = handR;
 4:   if (Lzwp.input.hands[i].leftHand)
 5:       tr = handL;
 6:
 7:   if (Lzwp.input.hands[i].pose.tracked)
 8:   {
 9:     if (moveHands)
10:     {
11:       tr.position = Lzwp.input.hands[i].pose.position;
12:       tr.rotation = Lzwp.input.hands[i].pose.rotation;
13:     }
14:
15:     for (int j = 0; j < 5; j++)  // for each finger
16:     {
17:       LzwpInput.FingertrackingFinger f = Lzwp.input.hands[i].fingers[j];
18:
19:       for (int k = 0; k < 3; k++)  // for each phalanx
20:       {
21:         Transform trP =
22:           GetFingerPhalanx(Lzwp.input.hands[i].rightHand, j, k);
23:
24:         trP.localScale = new Vector3(
25:           f.phalanxLength[k],
26:           f.tipRadius * 2f,
27:           f.tipRadius * 2f
28:         );
29:
30:         trP.localPosition = f.joints[k].positionInHandCoordSystem +
31:           Vector3.left * (f.phalanxLength[k] / 2f) + // 0.5 finger length
32:           Vector3.right * 0.06f;  // 0.5 palm length
33:
34:         if (k == 0)  // outermost phalanx
35:           trP.localRotation = f.tipPose.rotation;
36:         else if (k == 1)  // middle phalanx
37:           trP.localRotation = f.tipPose.rotation *
38:               Quaternion.Euler(0f, -f.phalanxAngle[0], 0f);
39:         else if (k == 2)  // innermost phalanx
40:           trP.localRotation = f.tipPose.rotation *
41:               Quaternion.Euler(0f, -f.phalanxAngle[0], 0f) *
42:               Quaternion.Euler(0f, -f.phalanxAngle[1], 0f);
43:       }
44:     }
45:   }
46:
47:   if (disableWhenNotTracked)
48:     tr.gameObject.SetActive(Lzwp.input.hands[i].pose.tracked);
49: }
```

1.17.6. *Śledzenie ciała*

Przykład *Mocap\Mocap_LzwpExample* demonstruje możliwość animacji humanoidalnego modelu na podstawie danych odbieranych z systemu śledzenia, a konkretnie z aplikacji ART-Human (rys. 1.13). Podobnie jak w przykładzie ze śledzeniem dłoni i palców, na scenie znajduje się obiekt reprezentujący średnią jaskinię LZWP. Ze względów licencyjnych, oraz aby nie zwiększać niepotrzebnie rozmiaru pakietu (śledzenie ciała jest stosunkowo rzadko wykorzystywane), scena nie zawiera natomiast modelu, który mógłby być animowany – aby skorzystać z tego przykładu, należy najpierw samodzielnie dodać odpowiedni model. Można w tym celu wykorzystać np. model Y lub X Bota (rys. 1.14), dostępny do pobrania za darmo z serwisu Mixamo [13].



Rys. 1.13. Przykład podpięcia systemu śledzenia ciała



Rys. 1.14. Y Bot dostępny w serwisie Mixamo (widok modelu oraz jego szkieletu)

Po zaimportowaniu modelu należy upewnić się, że jest on oznaczony jako humanoidalny (w oknie inspektora modelu na karcie *Rig* pole *Animation Type* ustawione jest na wartość *Humanoid*), oraz że odpowiednio zdefiniowany jest jego awatar – podpięte są pożądane elementy modelu (konfigurację można sprawdzić klikając przycisk *Configure Avatar* w oknie inspektora awatara lub

przycisk *Configure...* na karcie *Rig* inspektora modelu). Na koniec do zaimportowanego i skonfigurowanego modelu wstawionego na scenę dodać jeszcze należy skrypt *Mocap_AnimateModel.cs*. W oknie inspektora można ręcznie ustawić przypisanie poszczególnych kości, jednak zalecane jest korzystanie z przypisywania automatycznego, wykonywanego po uruchomieniu aplikacji w metodzie *Start*. Przypisanie automatyczne wymaga podpiętego do modelu komponentu *Animator*, w którym na odpowiedni obiekt ustawione jest pole *Avatar*. Za pomocą metody *GetBoneTransform* tego komponentu kolejne kości zapisywane są do tablicy *modelBoneTransforms*. W metodzie *Update* pobierane są z tablicy *Lzwp.input.humans* aktualne dane dla pierwszego modelu, a następnie rotacja każdego śledzonego obiektu z listy *joints* przypisywana jest do odpowiednio zmapowanej kości przechowywanej w tablicy *modelBoneTransforms*. Aby model mógł się przemieszczać, dla głównego elementu hierarchii szkieletu (biodra/kość miednicza) dodatkowo ustawiana jest także pozycja. Kod animujący model wygląda więc następująco:

```
LzwpInput.Human h = Lzwp.input.humans[0];

for (int i = 0; i < h.joints.Length && i < modelBoneTransforms.Length; i++)
{
    if (h.joints[i].tracked)
    {
        if (modelBoneTransforms[i] == null)
            continue;

        if (i == 0)  // hips/pelvis
            modelBoneTransforms[i].position = h.joints[0].position;

        modelBoneTransforms[i].rotation = h.joints[i].rotation;
    }
}
```

1.17.7. *Synchronizacja – pozycja, rotacja i skala obiektów*

Przykład *Sync\SyncTransform_LzwpExample* demonstruje synchronizację pozycji, rotacji i skali obiektu. Znajdujący się na scenie sześcian (obiekt *Cube*) przypięty ma komponent *NetworkView*, którego domyślna konfiguracja powoduje synchronizację tych właśnie właściwości. Obiekt sześcianu podpięty ma również skrypt *SyncTransform_SyncExample*, który w metodzie *Update* korzystając z pola *Lzwp.sync.isMaster* sprawdza najpierw, czy bieżąca instancja aplikacji działa w trybie serwera, i jeśli tak, to sukcesywnie modyfikuje wymienione właściwości obiektu. Istotne jest, aby modyfikacji takich dokonywała jedynie instancja serwerowa, a następnie stan obiektu przesyłała do klientów, którzy aktualizują na tej podstawie stan danego obiektu po swojej stronie. Zapewnia to spójność stanu aplikacji. Warto również zwrócić uwagę na pole *State Synchronization* komponentu *NetworkView*, określające sposób synchronizacji. Może ono przyjąć jedną z następujących wartości:

- *Off* (wartość 0) – synchronizacja wyłączona;
- *Reliable Delta Compressed* (wartość 1) – dane są wysyłane jedynie wtedy, gdy zaszła w nich zmiana od czasu ostatniej synchronizacji; Unity zapewnia dostarczenie wszystkich pakietów (ponowne wysyłanie w przypadku braku potwierdzenia odbioru), w odpowiedniej kolejności;

- *Unreliable* (wartość 2) – dane przesyłane są podczas każdego wywołania synchronizacji, niezależnie od tego, czy zostały zmodyfikowane; dostarczenie pakietu nie jest gwarantowane.

1.17.8. *Synchronizacja – RPC*

Przykład Sync*SyncRPC_LzwpExample* pokazuje jak użyć można metody *RPC* obiektu *NetworkView*. Na scenie znajduje się sześcian (obiekt *Cube*). Przypięty jest do niego komponent *NetworkView*, w którym wyłączona została synchronizacja stanu, gdyż nie będzie ona wykorzystywana. Referencja do tego obiektu przypisana zostaje do zmiennej *nv* w metodzie *Start* skryptu *ChangeProperties_SyncExample*, który również przypięty jest do sześcianu. Skrypt ten umożliwia zmianę koloru oraz skali kostki na kolejne wartości losowe za pośrednictwem metod *ChangeColor* i *ChangeSize*, które wywoływane są wciśnięciem odpowiednio pierwszego i drugiego od lewej przycisku kontrolera Flystick (lub alternatywnie klawisza 1 i 2 na klawiaturze). Zmiany te inicjowane są po stronie aplikacji serwerowej (tryb działania sprawdzany jest za pomocą pola *Lzwp.sync.isMaster*) i synchronizowane z wszystkimi instancjami klienckimi. Zastosowano tutaj dwa nieco różniące się podejścia:

- zmiana rozmiaru – w metodzie *ChangeSize* (wywoływanej wyłącznie na serwerze) losowany jest nowy rozmiar kostki, zmieniana jest z jego wykorzystaniem skala obiektu po stronie serwerowej, a następnie na obiekcie *NetworkView* (zmienna *nv*) wywoływana jest metoda *RPC*. Powoduje ona wywołanie w pozostałych, klienckich instancjach aplikacji (*RPCMode.OthersBuffered*), metody wskazanej jako pierwszy parametr *RPC*, czyli metody *ChangeSizeRPC*. Dodatkowo przekazywany jest też wylosowany wcześniej rozmiar (parametr typu *int*), na podstawie którego wywołana na klientach metoda aktualizuje skalę wyświetlanej przez nie kostki.

```
void ChangeSize()
{
    float s = Random.Range(0.6f, 2f);
    transform.localScale = Vector3.one * s;
    nv.RPC("ChangeSizeRPC", RPCMode.OthersBuffered, s);
}

[RPC]
void ChangeSizeRPC(float scale)
{
    transform.localScale = Vector3.one * scale;
}
```

- zmiana koloru – w metodzie *ChangeColor* (wywoływanej wyłącznie na serwerze) losowany jest nowy kolor sześcianu, jednak – inaczej niż w przypadku rozmiaru – nie jest on od razu przypisywany do kostki instancji serwerowej. Poprzez RPC wywoływana jest natomiast metoda *ChangeColorRPC* – tym razem wywołanie to dotyczy wszystkich węzłów, również serwera (*RPCMode.AllBuffered*). Jako 4 ostatnie parametry przekazywany jest wylosowany wcześniej kolor, rozbity na 4 jego składowe typu *float* (kolor czerwony, zielony, niebieski i wartość nieprzezroczystości), ponieważ zgodnie z dokumentacją [11] typ *Color* nie może zostać przekazany bezpośrednio (nie ma go na liście dozwolonych typów). Wywołana na każdej instancji

metoda *ChangeColorRPC* zmienia kolor kostki (przypisując go do materiału, do którego referencję przypisano wcześniej do zmiennej *mat*).

```
void ChangeColor()
{
    Color c = Random.ColorHSV(0, 1f, 0.8f, 1f, 0.8f, 1f, 1f, 1f);
    nv.RPC("ChangeColorRPC", RPCMode.AllBuffered, c.r, c.g, c.b, c.a);
}

[RPC]
void ChangeColorRPC(float r, float g, float b, float a)
{
    mat.color = new Color(r, g, b, a);
}
```

Warto zwrócić uwagę, że metody, które mają być wywoływane za pośrednictwem mechanizmu wywołania zdalnego, muszą zostać oznaczone atrybutem *RPC*.

W parametrze określającym węzły, na których ma nastąpić wywołanie metody, użyć można również wartości *All* i *Others* – odpowiadają one wartościom *AllBuffered* i *OthersBuffered*, jednak ich wywołania nie są buforowane i wywołanie metod nie zostanie nadrobione przez instancje, które podłączą się do serwera już po wywołaniu przez niego metody *RPC*. Oczywiście buforowanie takie nie zawsze jest pożądane, np. metoda inicjująca animację wybuchu prawdopodobnie nie powinna już być wywoływana w instancjach uruchomionych z opóźnieniem.

1.17.9. *Synchronizacja – tworzenie i usuwanie obiektów*

Przykład *Sync\SyncInstantiation_LzwpExample* demonstruje możliwość synchronizowanego tworzenia i usuwania obiektów. Wzorcem takiego obiektu będzie prefabrykat *cubeObject*, zawierający model sześcianu. Cztery tego typu obiekty zostały ręcznie umieszczone na scenie wewnątrz kontenera *Cubes wrapper* (aby na początku działania aplikacji były jakieś obiekty możliwe do usunięcia). Prefabrykat *cubeObject* ma podpięty skrypt *InstantiatedCube_SyncExample*, który zawiera publiczną metodę *InitCube* – jej wywołanie w instancji serwerowej spowoduje przypisanie danej kostce koloru, przekazanego jako parametr tej metody, oraz nazwy „cubeObject” – usuwany jest przyrostek „(Clone)”, który otrzymuje nazwa obiektu utworzonego w trakcie działania aplikacji. Następuje też przypisanie jej do rodzica, wskazanego w polu *cubesWapper* skryptu *Instantiation_LzwpExample* (będzie to wspomniany już kontener *Cubes wrapper*).

Do obiektu podążającego za pierwszym kontrolerem typu Flystick podpięto półprzezroczystą, niebieską kostkę (obiekt *CubeIndicator*) – będzie ona wskazywała kolor i miejsce utworzenia kolejnej kostki z prefabrykatu – oraz małą, czerwoną sferę (*SphereIndicator*), która będzie pozwalała usuwać obiekty (klony prefabrykatu) z nią kolidujące. Obiekt sfery podpięty ma skrypt *IndicatorRemoving_SyncExample*, którego zadaniem jest aktualizacja w instancji serwerowej listy *elementsToDestroy* ze skryptu *Instantiation_LzwpExample*. Obiekty *cubeObject* wchodzące w kolizję z tą sferą dodawane są do listy za pomocą metody *AddElementToDestroyList*, natomiast wychodzące z kolizji – usuwane są z listy metodą *RemoveElementFromDestroyList*.

Na scenie znajduje się również obiekt *GameManager*, do któego podpięty jest główny skrypt tego przykładu – *Instantiation_LzwpExample* – oraz skrypt pomocniczy – *ToggleObject_SyncExample*. Zadaniem skryptu pomocniczego jest przełączanie widoczności obiektu *CubeIndicator* poprzez metodę *ToggleVisibility* oraz losowa zmiana jego orientacji poprzez metodę *ChangeOrientation*. Pierwsza z tych metod wywoływana jest trzecim od lewej przyciskiem kontrolera Flystick lub klawiszem z numerem 3 na klawiaturze, druga zaś – przyciskiem czwartym od lewej lub klawiszem z numerem 4.

Skrypt *Instantiation_LzwpExample* pozwala przełączać tryb pracy pomiędzy dodawaniem a usuwaniem obiektów – klonów prefabrykatu *cubeObject*. Z użyciem metody *UpdateIndicators* przełącza widoczność obiektów *CubeIndicator* i *SphereIndicator* w zależności od aktualnego trybu działania. Tryb dodawania włączyć można przy użyciu pierwszego od lewej przycisku kontrolera Flystick (lub alternatywnie klawiszem z numerem 1 na klawiaturze), natomiast tryb usuwania – drugim od lewej przyciskiem kontrolera (alternatywnie: klawiszem z numerem 2). Spust kontrolera i klawisz spacji umożliwiają wykonanie wybranej akcji, które zaimplementowane jest w metodzie *ExecuteAction*, przedstawionej na listingu poniżej:

```
void ExecuteAction()
{
    if (adding)
    {
        GameObject cube = Network.Instantiate(cubePrefab,
            indicatorAdding.transform.position,
            indicatorAdding.transform.rotation, 0) as GameObject;
        cube.GetComponent<InstantiatedCube_SyncExample>().InitCube(mat.color);
        SetRandomIndicatorColor();
    }
    else
    {
        foreach (GameObject go in elementsToDestroy)
            Network.Destroy(go);
        elementsToDestroy.Clear();
    }
}
```

Utworzenie klonu prefabrykatu realizowane jest z wykorzystaniem statycznej metody *Instantiate* klasy *Network*. Jako parametry przekazywane są jej: wzorcowy prefabrykat (przypisany wcześniej do zmiennej *cubePrefab*), pozycja i rotacja pobrane z obiektu *CubeIndicator* (zmienna *indicatorAdding*), oraz grupa sieciowa – koniecznie ta oznaczona numerem 0. Metoda zwraca referencję do nowo utworzonego obiektu, w którego skrypcie *InstantiatedCube_SyncExample* wywoływana jest następnie metoda *InitCube* z odpowiednim kolorem przekazanym jak parametr. Na początku działania aplikacji oraz po utworzeniu nowego obiektu pomocnicza metoda *SetRandomIndicatorColor* przypisuje obiektowi *CubeIndicator* nowy, losowy kolor, który zostanie wykorzystany przy tworzeniu kolejnego sześcianu (zmienna *mat* odnosi się do materiału obiektu *CubeIndicator*). Wywołanie akcji (metody *ExecuteAction*) w trybie usuwania obiektów powoduje natomiast wywołanie (tylko w instancji serwerowej) dla każdego obiektu z listy *elementsToDestroy* statycznej metody *Destroy* klasy *Network*, gdzie obiekt ten przekazywany jest jako parametr.

Następnie lista jest czyszczona. Użycie wskazanej metody powoduje usunięcie danego obiektu w instancji serwerowej, a także odpowiadających mu obiektów w instancjach klienckich.

1.17.10. *Synchronizacja – czas*

Przykład *Sync time\SyncTime_LzwpExample* pokazuje możliwość odczytu zsynchronizowanego czasu. W skrypcie *Timer_SyncTimeExample*, przypiętym do obiektu *GuiTimer*, w metodzie *OnGUI* rysowana jest etykieta (wyświetlana w lewym górnym rogu ekranu) z tekstem „Synchronized time: ” oraz czasem w sekundach, pobranym za pomocą pola *Lzwp.sync.time*:

```
if (Lzwp.initialized)
    GUILayout.Label("Synchronized time: " + Lzwp.sync.time);
```

Na scenie znajduje się również obiekt *Quad* z materiałem używającym programu cieniującego *TextureAnimationExample*. Na materiale tym wyświetlana jest tekstura czarno-białej siatki, której prosta animacja – przesuwanie w obu osiach – zrealizowana jest w programie cieniującym poprzez ustawianie współrzędnych UV na wartości zależne od synchronizowanego czasu, pobrane z wektorów *_LzwpTime* oraz *_LzwpSinTime* i przemnożone przez wartości z wektora *_AnimationSpeed*, określającego prędkość animacji. Deklaracje zmiennych dotyczących czasu zostały wcześniej dołączone do kodu programu cieniującego przy użyciu dyrektywy *include*:

```
// include declarations: _LzwpTime, _LzwpSinTime, _LzwpCosTime
#include "Assets/LZWPlib/Shaders/LZWPlibCG.cginc"

// ...
v2f vert (appdata v)
{
    v2f o;
    //...

    // shift the UVs over time
    o.uv.x += _AnimationSpeed.x * _LzwpTime.x;
    o.uv.y += _AnimationSpeed.y * _LzwpSinTime.x;
    return o;
}
```

6.1.17.11. *Synchronizacja – przełączanie scen*

Przykład *Switch scene\LzwpExampleSceneX* zawiera dwie sceny: *LzwpExampleScene1* oraz *LzwpExampleScene2*. Wymaga on wstępnego przygotowania – dodania owych scen do listy wykorzystywanych scen w ustawieniach budowania projektu. Aby dodać te sceny do listy, użyć można przycisku *Prepare example*, aby je natomiast stamtąd usunąć – przycisku *Clear example*. Obie sceny są bardzo do siebie podobne. Różnią się tym, że pierwsza z nich zawiera jeden obracający się (za pośrednictwem skryptu *RotateObject_LzwpExample*) fioletowy sześcian, natomiast druga – dwa obracające się sześciany niebieskie. Mają one służyć do odróżniania tych scen od siebie oraz do pokazania, że po załadowaniu sceny prawidłowo działa synchronizacja stanu obiektów. Na obu scenach znajdują się również obiekty *GameManger*, do których podpięty jest skrypt *SwitchSceneExample*. Umożliwia on:

- załadowanie sceny pierwszej lub drugiej – do tego celu użyto metodę *Lzwp.sync.LoadScene*, do której przekazywana jest nazwa odpowiedniej sceny (ich nazwy przechowywane są w stałych *SCENE_1_NAME* i *SCENE_2_NAME*); załadownie sceny pierwszej następuje po wciśnięciu pierwszego od lewej przycisku dowolnego kontrolera Flystick lub klawisza z numerem 1 na klawiaturze, do załadowania sceny drugiej służy natomiast drugi od lewej przycisk kontrolera oraz klawisz z numerem 2; wybranie załadowania sceny, która jest już w danym momencie załadowana, spowoduje jej przeładowanie;
- przeładowanie bieżącej sceny – w tym celu wykorzystano metodę *Lzwp.sync.ReloadScene*; ponowne załadowanie sceny następuje po wciśnięciu czwartego od lewej przycisku dowolnego kontrolera Flystick lub klawisza R na klawiaturze;
- naprzemienne przełączanie się pomiędzy obiema scenami – zrealizowane w metodzie *ToggleScene*, gdzie najpierw sprawdzana jest nazwa obecnie załadowanej sceny – *Lzwp.sync.GetActiveScene().name* – a następnie druga ze scen ładowana jest z użyciem metody *Lzwp.sync.LoadScene*; przełączenie sceny wywoływane jest wciśnięciem spustu dowolnego kontrolera Flystick lub klawisza spacji na klawiaturze.

2. Laboratorium Zanurzonej Wizualizacji Przestrzennej

5 grudnia 2014 r. na Politechnice Gdańskiej uruchomiono Laboratorium Zanurzonej Wizualizacji Przestrzennej (LZWP), będące częścią projektu „Nowoczesne Audytoria Politechniki Gdańskiej". Mieści się ono w specjalnie dobudowanym w tym celu budynku (rys. 2.1), znajdującym się na tyłach starego gmachu Wydziału Elektroniki, Telekomunikacji i Informatyki. Główne pomieszczenie jest wyłożonym czernią sześcianem o boku długości 12,5 m, w którego środku znajduje się kluczowa instalacja laboratorium – jak dotąd jedyna w Polsce pełna jaskinia rzeczywistości wirtualnej, BigCAVE. Około 1,5 roku po otwarciu LZWP jego wyposażenie zostało wzbogacone o dwie dodatkowe jaskinie: średnią – MidiCAVE – złożoną z trzech ścian i podłogi oraz małą – MiniCAVE – zbudowaną z 4 monitorów stereoskopowych o analogicznym do jaskini średniej układzie. Dodano też stanowiska deweloperskie, stanowiące pierwszy krok procesu wytwarzania i testowania aplikacji dla jaskiń rzeczywistości wirtualnej. Wszystkie te instalacje mają zbliżone parametry konstrukcyjne, dzięki czemu testowanie aplikacji w dużym, zaawansowanym systemie może być poprzedzone testami na systemie prostszym i tańszym w eksploatacji. Laboratorium dysponuje także sferycznym symulatorem chodu (ang. *spherical walk simulator*), który może być używany w parze z kaskiem rzeczywistości wirtualnej, jednak może być również wprowadzony do wnętrza dużej jaskini, co jest rozwiązaniem unikatowym na skalę światową.

Rys. 2.1. Budynek Laboratorium Zanurzonej Wizualizacji Przestrzennej

W kolejnych podrozdziałach szczegółowo opisane zostaną poszczególne stanowiska laboratoryjne, jednak najpierw przedstawione będą dwa aspekty wspólne dla tych stanowisk.

2.1. Połączenia sieciowe

Wszystkie komputery w laboratorium wchodzące w skład jaskiń rzeczywistości wirtualnej połączone są ze sobą za pomocą dwóch alternatywnych sieci: gigabitowego łącza Ethernet oraz sieci InfiniBand o przepustowości 40 Gb/s. Dla poszczególnych jaskiń wyodrębnione zostały sieci logiczne (VLAN , ang. *Virtual Local Area Network*), aby dane przeznaczone dla jednej z nich nie zakłócały pracy innej. Koncepcja ta została jednak w ostatnim czasie zarzucona, aby m.in. umożliwić aplikacjom wykorzystanie mocy obliczeniowej większej liczby komputerów oraz aby powstawać mogły współpracujące ze sobą aplikacje działające na kilku stanowiskach (np. aplikacja z grą w baseballa lub symulator naprowadzania śmigłowca na lądowisko, które działały w połączeniu jaskini małej ze średnią). W przypadku poszczególnych jaskiń bezpośredni dostęp możliwy jest jedynie do jednego, głównego komputera. Dostęp do pozostałych węzłów uzyskać można poprzez połączenie pulpitu zdalnego z wykorzystaniem aplikacji TightVNC lub korzystając z cienkich klientów protokołu PCoIP (ang. *PC-over-IP*).

Dla LZWP przygotowane zostało również połączenie ze specjalnie wydzielonym segmentem superkomputera TRYTON, który znajduje się w pobliskim Centrum Informatycznym TASK. Udostępniono środowisko zwirtualizowane działające na 11 węzłach, z których każdy działa pod kontrolą systemu CentOS i posiada 40 procesorów o taktowaniu 2,3 GHz oraz 100 GB pamięci RAM [14]. Połączenie to ma umożliwić realizację dwóch podstawowych scenariuszy – pierwszy zakłada zlecanie wykonania obliczeń (np. symulacji fizycznych) i zwracanie wyników, które następnie będą dalej przetwarzane po stronie laboratorium; drugi zaś polega na przesłaniu do superkomputera danych, na podstawie których będzie mógł on wyrenderować i zwrócić obraz, który zostanie bezpośrednio wyświetlony na ścianach jaskini [15].

Komputery laboratoryjne, które służą do przygotowywania i początkowego testowania aplikacji, mają dostęp do sieci Internet. Znacząco ułatwia to proces adaptacji aplikacji. Są to jednostki na stanowiskach deweloperskich oraz główny węzeł małej jaskini. Pozostałe węzły – ze względów bezpieczeństwa – takiego dostępu nie posiadają.

2.2. Systemy śledzące

W Laboratorium Zanurzonej Wizualizacji Przestrzennej do obsługi śledzenia wykorzystywane są systemy optyczne niemieckiej firmy ART Advanced Realtime Tracking GmbH [16]. System taki wyposażony jest w odpowiednio skalibrowane ze sobą kamery śledzące (co najmniej dwie, o pokrywającym się częściowo polu widzenia) oraz jednostkę przetwarzającą i udostępniającą dane. Kamery otoczone są diodami LED synchronicznie błyskającymi światłem podczerwonym. Światło to odbija się od odblaskowych markerów (są też markery aktywne, emitujące własne światło podczerwone) i zostaje zarejestrowane przez kamery, które dokonują przetwarzania wstępnego – na widzianym przez siebie obrazie w skali szarości metodą rozpoznawania wzorców wyznaczają położenie markerów ze średnią dokładnością 0,04 px przy rozdzielczości obrazu 1,1 – 1,3 Mpx. Wyniki tego procesu przesyłają łączem Ethernet do jednostki głównej (wyjątek stanowi system SMARTTRACK, gdzie kamery są zintegrowane z układem jednostki głównej). Jednostka ta wylicza

miejsca przecięć promieni wychodzących z poszczególnych kamer i przechodzących przez zidentyfikowane przez nie punkty – przecięcia te określają położenie markerów (3 DoF) w prawoskrętnym układzie współrzędnych systemu śledzenia. Pojedyncze markery można łączyć w grupy. Unikatowy, stały układ co najmniej 4 markerów – zwany dalej targetem – może zostać zapamiętany przez jednostkę główną podczas procesu jego kalibracji.

Wówczas wyliczana będzie pozycja ustalonego punktu, reprezentującego dany target, oraz orientacja tego targetu, co łącznie daje 6 stopni swobody (6 DoF). Punkt reprezentujący target – będący początkiem jego układu współrzędnych – oraz kierunek osi tego układu wyznaczane są następująco:

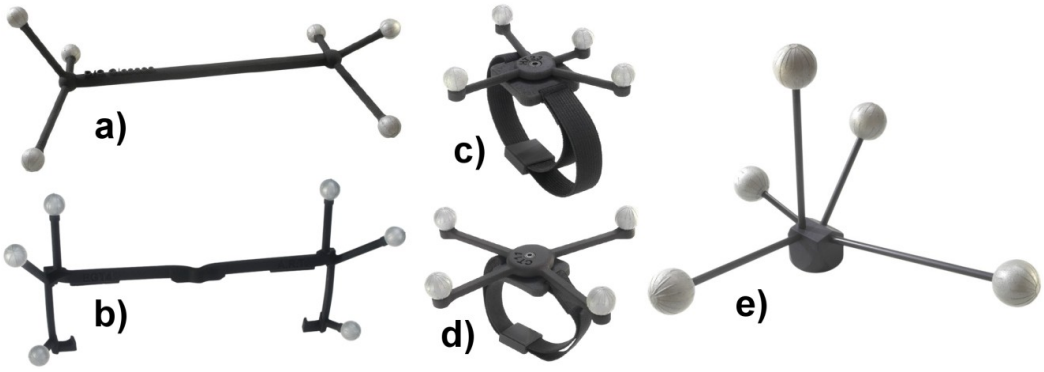
1. Znajdź dwa najbardziej oddalone od siebie markery (*A* i *B*, przypisanie liter zostanie określone w kolejnym kroku).
2. Znajdź trzeci marker (*C*), który będzie w najmniejszej odległości od jednego z dwóch znalezionych w poprzednim kroku. Marker z pary {*A*, *B*}, który jest bliżej markera *C*, jest markerem *A*.
3. Marker *A* jest początkiem układu współrzędnych targetu. Dodatnia półoś *X* biegnie od markera *A* przez marker *B*. Marker *C* wraz z dwoma poprzednimi wyznacza płaszczyznę *XY*, na której znajdują się osie *X* i *Y*. Jego współrzędna *y* jest dodatnia – leży on na półpłaszczyźnie, na której prostopadła do osi *X* półoś *Y* ma wartości dodatnie. Oś *Z* wyznaczyć można na podstawie prawoskrętności układu współrzędnych.

Firma ART oferuje predefiniowane zestawy targetów, możliwe jest również utworzenie i skalibrowanie własnych układów [17]. Laboratorium dysponuje różnego rodzaju targetami – ich zestawienie wraz z wykorzystaniem przy poszczególnych stanowiskach przedstawia tabela 2.1, natomiast na rysunku 2.2 przedstawiono wygląd niektórych z nich. Targety w tej samej grupie mają identyczny układ markerów. Skrótem MC oznaczono zestaw do śledzenia ciała (ang. *Motion Capture*), natomiast skrótem FT – do śledzenia dłoni i palców (ang. *Finger Tracking*). Zestawy te przedstawiono na rys. 2.3. Niewykorzystane targety mogą zostać użyte do śledzenia dowolnego obiektu, do którego zostaną przyczepione. Aby dany target był śledzony, co najmniej 4 jego markery muszą być widoczne przez co najmniej dwie kamery śledzące. Użytkownik musi więc uważać, aby nie wyjść z targetem poza zasięg kamer oraz aby nie zasłaniać go nadmiernie np. własnym ciałem.

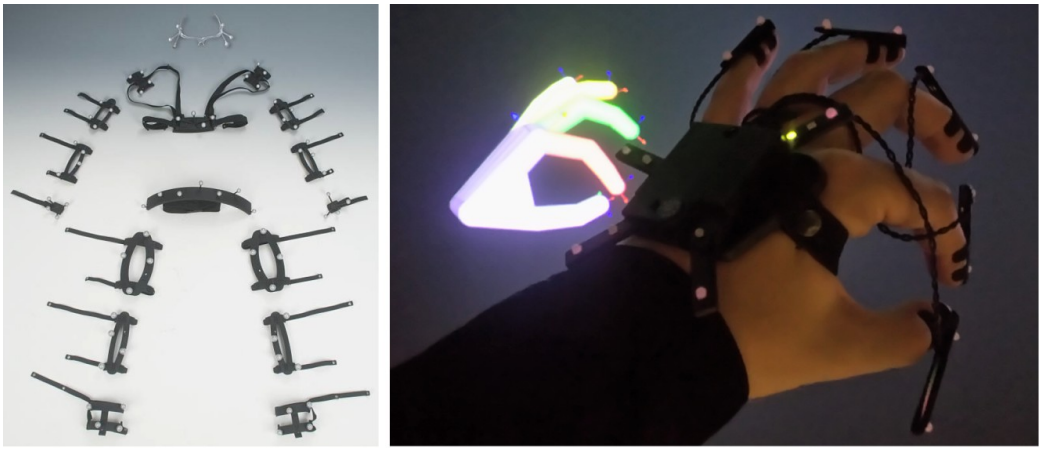
Tabela 2.1. Wykorzystanie targetów przy poszczególnych stanowiskach

Symbol	Opis	Grupa	Zestaw	Liczba sztuk				
				MiniCAVE	MidiCAVE	BigCAVE	Niewykorzystane	Razem
GT4	okulary (target uniwersalny)	4				1	5	6
IPT4	okulary (ramka Infitec)	4			1		2	3
3dVT4	okulary (ramka 3D Vision)	4					1	1
AG2T4	ramka okularów (brak szkieł)	4	MC		1			1
GT5	okulary (target uniwersalny)	5			1	1	1	3
3dVT5	okulary (ramka 3D Vision)	5					1	1
GT6	okulary (target uniwersalny)	6					3	3
C3dT7	okulary (ramka Cinema3D)	7					2	2
EGT8	okulary (ramka Volfoni EDGE)	8		1			1	2
RCS410	zestaw do kalibracji			1	1	1		3
TT01	Tree Target						1	1
TT02	Tree Target						1	1
TT03	Tree Target						1	1
CT11	Claw Target						1	1
CT12	Claw Target						1	1
CT13	Claw Target						1	1
CT14	Claw Target						1	1
HT23	Hand Target		MC		1		2	3
HT25	Hand Target		MC		1		1	2
HT26	Hand Target						1	1
HT27	Hand Target						1	1
DT1	Back Target 1 (plecy/brzuch)		MC		1			1

WT1	Waist Target 1 (talia przód)		MC		1			1
WT2	Waist Target 2 (talia tył)		MC		1			1
UT1	Shoulder Target 1 (bark lewy)		MC		1			1
UT2	Shoulder Target 2 (bark prawy)		MC		1			1
HBT1	Upper Arm Target 1 (ramię lewe)		MC		1			1
HBT2	Upper Arm Target 2 (ramię prawe)		MC		1			1
UBT1	Forearm Target 1 (przedramię lewe)		MC		1			1
UBT2	Forearm Target 2 (przedramię prawe)		MC		1			1
FBT1	Thigh Target 1 (udo lewe)		MC		1			1
FBT2	Thigh Target 2 (udo prawe)		MC		1			1
TBT1	Lower Leg Target 1 (łydka lewa)		MC		1			1
TBT2	Lower Leg Target 2 (łydka prawa)		MC		1			1
FT1	Foot Target 1 (stopa lewa)		MC		1			1
FT2	Foot Target 2 (stopa prawa)		MC		1			1
H1L	Fingertracking (dłoń lewa)		FT		1			1
H2R	Fingertracking (dłoń prawa)		FT		1			1
			Razem	2	23	3	28	56



Rys. 2.2. Predefiniowane targety [16]: a) uniwersalny okularowy, b) ramka na okulary Volfoni EDGE, c) Hand Target, d) Claw Target, e) Tree Target



Rys. 2.3. Targety zestawu do śledzenia ciała [16] (po lewej) oraz do śledzenia dłoni i palców (po prawej)

LZWP posiada również kilka kontrolerów typu Flystick 2 (rys. 2.4) od firmy ART, które dają użytkownikowi w jaskini możliwość interakcji z aplikacjami. Ich zestawienie i przypisanie do poszczególnych jaskiń przedstawiono w tabeli 2.2. Kontrolery w tej samej grupie mają identyczny układ markerów. Flystick wyposażony jest w zestaw ukrytych markerów oraz analogowy joystick i 6 przycisków – spust, 4 przyciski pod joystickiem, przycisk w joysticku. Jednostka główna oprócz śledzenia pozycji i orientacji tych kontrolerów łączy się z nimi drogą radiową (2,4 GHz) i tym kanałem pobiera stan przycisków i wychylenia joysticków. Jednocześnie obsługiwanych może być do 5 kontrolerów Flystick 2.

Rys. 2.4. Kontroler Flystick 2

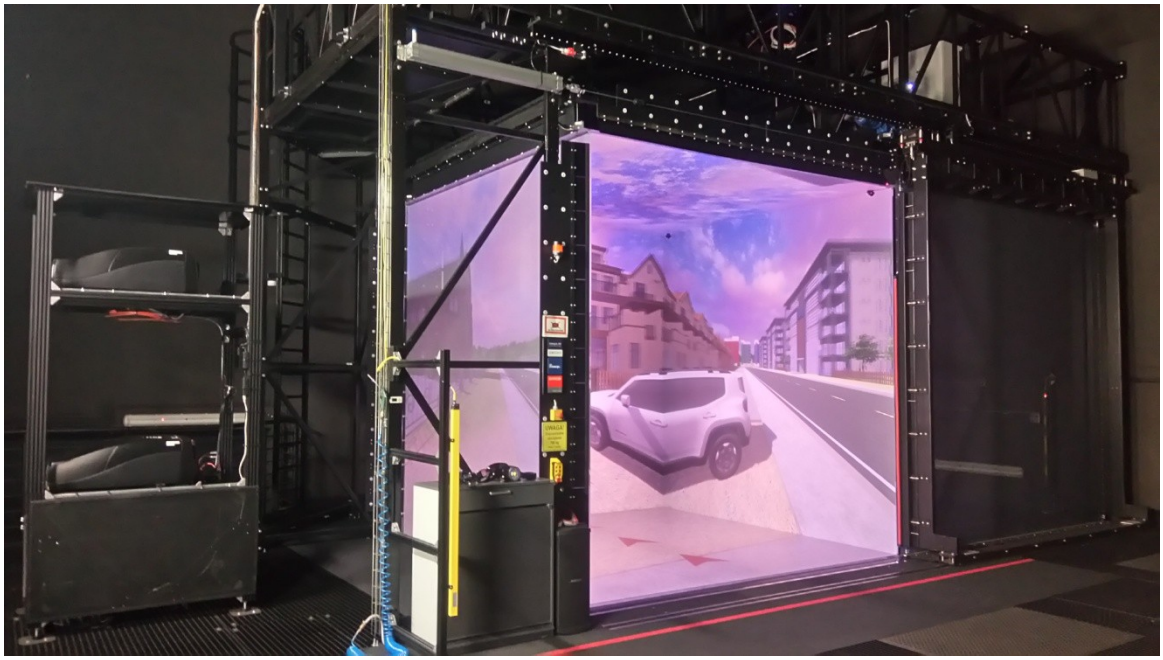
Tabela 2.2. Przypisanie kontrolerów Flystick do poszczególnych stanowisk

Numer seryjny	Grupa	Stanowisko i slot
00428	1	MidiCAVE 1
00445	2	MiniCAVE 3, MidiCAVE 4
00601	1	---
00602	1	BigCAVE 1
00608	2	BigCAVE 2
00611		BigCAVE 3, MidiCAVE 3
00612		MiniCAVE 2
00613		MidiCAVE 2
00614	1	MiniCAVE 1

Jednostka główna przesyła zebrane dane (pozycje i orientacje wszystkich obiektów, stan przycisków i joysticków kontrolerów) w formie tekstowej z maksymalną częstotliwością 60 Hz na wskazane w jej konfiguracji adresy IP i numery portów, korzystając z protokołu UDP.

2.3. BigCAVE

Pierwszą i największą instalacją VR, jaka powstała w LZWP, jest duża jaskinia rzeczywistości wirtualnej – BigCAVE (rys. 2.5). Ma ona formę sześcianu o boku długości 3,4 m, składa się z 6 kwadratowych ekranów akrylowych – 4 ścian, sufitu oraz podłogi (która dodatkowo wzmocniona jest od dołu warstwą szkła). Jeden z ekranów – ściana tylna – posiada mechanizm przesuwający go; po odsunięciu umożliwia wejście do jaskini. Po zamknięciu dodatkowo wrota te dociskane są do pozostałych ścian, aby zminimalizować szczeliny na łączeniu ekranów. Ekrany napylone mają od wewnątrz warstwę projekcyjną. Obraz rzucany jest na nie od zewnątrz przez 12 projektorów – po dwa projektory na ekran. W przypadku projekcji sufitowej i podłogowej użyte zostały lustra odbijające obraz w górę lub w dół. Rozdzielczość rzucanego obrazu wynosi 1920×1200 px, natomiast wynikowa rozdzielczość pojedynczego ekranu to 1920×1920 px. Obraz z pary projektorów częściowo pokrywa się w środkowej części danego ekranu, tworząc pas tzw. blendingu, dzięki któremu zaciera się granica łączenia tego obrazu. Obszar nakładania się stanowi ¼ ekranu (pas ma wymiary 0,85×3,4 m). Zachodzenie na siebie obrazu jest nieco mniejsze na podłodze, ze względu na belkę podtrzymującą konstrukcję, jednak parametry projekcji są analogiczne jak w przypadku pozostałych ekranów. Częstotliwość odświeżania obrazu to 120 Hz. Zastosowane zostały dwie działające jednocześnie technologie uzyskiwania stereoskopii: pasywna z rozdziałem widma oraz aktywna z wykorzystaniem okularów migawkowych.



Rys. 2.5. Duża jaskinia rzeczywistości wirtualnej – BigCAVE

Każdy z 12 projektorów podłączony jest do oddzielnego komputera, który renderuje dla niego obraz. Węzły te zamontowane są w dwóch szafach rack w pomieszczeniu serwerowym. Przypisano im adresy IP z przedziału 192.168.11.1 – 192.168.11.12 (tabela 2.3). Trzynasty, główny węzeł jaskini, pełniący rolę serwera aplikacji, znajduje się w tzw. sterowni – pomieszczeniu przyległym do hali z BigCAVEm. Renderuje on obraz widziany z perspektywy użytkownika z okularami wiodącymi. Jest on wyświetlany na monitorze w sterowni, a czasami dodatkowo transmitowany jest na Audytorium nr 2 budynku WETI A (przyległe do laboratorium), gdzie prezentowany może być szerszemu gronu odbiorców w technologii stereoskopii pasywnej.

Każdy z węzłów dużej jaskini jest komputerem Dell Precision Tower 3610, pracującym pod kontrolą systemu Windows 7 (z możliwością przełączenia na system CentOS). Jednostki te wyposażone są w karty graficzne Nvidia Quadro K5000 i procesory Intel Xeon E5-1620 v2, mają po 32 GB pamięci RAM.

Dźwięk dostarcza 8 głośników umiejscowionych parami w górnych narożnikach jaskini oraz 7 głośników poza nią, w tym jeden superniskotonowy. Źródłem dźwięku jest główny komputer jaskini (CAVE-CTRL). Na komputerze AUDIO-CTRL znajduje się oprogramowanie pozwalające zarządzać konfiguracją urządzeń dźwiękowych. Oba ta komputery zamontowane są w szafie rack stojącej w sterowni. W pokoju tym znajduje się również panel z mikrofonem, dzięki któremu można porozumiewać się m.in. z osobami zamkniętymi w jaskini. Komunikacja w drugą stronę możliwa jest poprzez mikrofon z mikroportem. Dźwięk z tego mikrofonu również może być transmitowany na wspomniane wcześniej Audytorium nr 2. Komputer AUDIO-CTRL umożliwia także podgląd kamer monitoringu. Dwie takie kamery zamontowane są w przeciwległych, górnych narożnikach BigCAVE'a.

Śledzenie w dużej jaskini zapewnia jednostka główna systemu ART wraz z 4 kamerami umieszczonymi w rogach jaskini pod sufitem. System pozwala na powiązanie z nim do 20 kontrolerów

typu Flystick. Śledzić może maksymalnie 50 obiektów (wliczając w to kontrolery). Dane z systemu śledzenia przesyłane są do głównego węzła jaskini na port 5000.

Zgodnie z regulaminem [18] wewnątrz jaskini przebywać może maksymalnie 7 osób (o łącznej wadze nieprzekraczającej 700 kg), zaś na podeście głównej hali (wraz z jaskinią) – maksymalnie 10 osób. Można więc przyjąć, że przy otwartych wrotach jaskini w symulacji/pokazie na żywo uczestniczyć może maksymalnie 10 osób.

Tabela 2.3. Przypisanie adresów sieciowych do urządzeń dużej jaskini

Nazwa hosta	**Adres IP**	**Opis**
operator	192.168.11.100	Komputer główny (CAVE-CTRL)
	192.168.11.101	System śledzenia
Front-1	192.168.11.1	Komputer dla ekranu frontowego – części górnej
Front-2	192.168.11.2	Komputer dla ekranu frontowego – części dolnej
Right-1	192.168.11.3	Komputer dla ekranu prawego – części górnej
Right-2	192.168.11.4	Komputer dla ekranu prawego – części dolnej
Left-1	192.168.11.5	Komputer dla ekranu lewego – części górnej
Left-2	192.168.11.6	Komputer dla ekranu lewego – części dolnej
Bottom-1	192.168.11.7	Komputer dla ekranu dolnego – części lewej
Bottom-2	192.168.11.8	Komputer dla ekranu dolnego – części prawej
Top-1	192.168.11.9	Komputer dla ekranu górnego – części lewej
Top-2	192.168.11.10	Komputer dla ekranu górnego – części prawej
Door-1	192.168.11.11	Komputer dla ekranu tylnego – części górnej
Door-2	192.168.11.12	Komputer dla ekranu tylnego – części dolnej

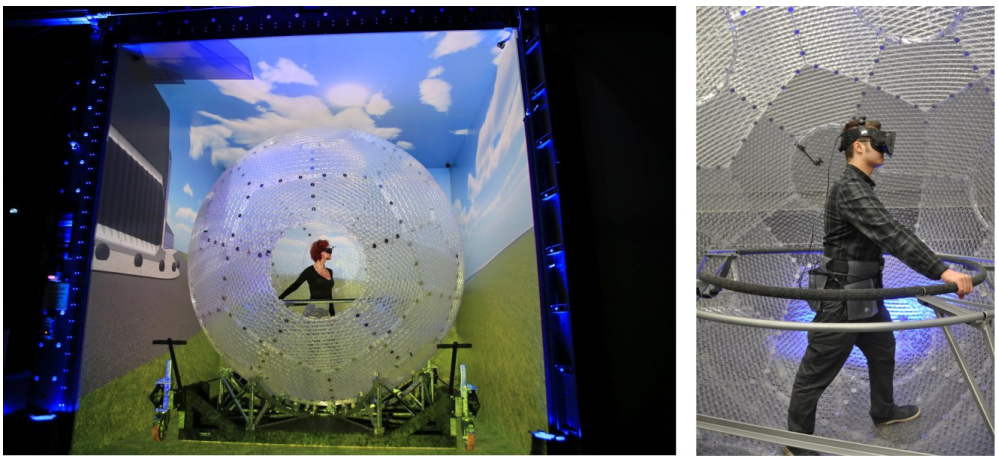
2.4. Sferyczny symulator chodu - Virtusphere

Sferyczny symulator chodu – urządzenie Virtusphere – jest ażurową sferą o średnicy 3,05 m i wadze ok. 390 kg [19], wykonaną z przezroczystego tworzywa sztucznego. Sfera posiada dwa włazy, po otwarciu których można wejść do jej wnętrza. Zgodnie z regulaminem [20] w danym momencie w środku może przebywać tylko 1 osoba (do wagi 110 kg). Sfera pozwala na stosunkowo naturalny chód bądź bieg, nieograniczony terytorialnie, który przenoszony jest do świata wirtualnego. W ten sposób można przemierzać wirtualne światy bez obaw o kolizję ze ścianą lub inną przeszkodą rzeczywistą. Dla początkujących użytkowników sfery dostępny jest w jej wnętrzu „chodzik” – metalowy okrąg na kółkach, którego można się chwycić stawiając pierwsze kroki.

Sferyczny symulator chodu może zostać użyty w sposób dwojaki (rys. 2.6):

1. dzięki specjalnie skonstruowanemu na potrzeby LZWP wózkowi, na którym osadzona jest sfera, symulator ten można wprowadzić do wnętrza dużej jaskini. Użytkownik widzi wówczas obraz wyświetlany na ścianach CAVE'a. Dla uzyskania stereoskopii i dopasowania perspektywy założyć może okulary standardowo używane w tej jaskini. Może również zabrać ze sobą kontroler Flystick, jeśli ma on zastosowanie w aplikacji. Obracająca się sfera sprawia wrażenie cienkiej firanki.
2. poza jaskinią z kaskiem rzeczywistości wirtualnej, który odpowiada za wyświetlanie obrazu i śledzenie głowy (przynajmniej jej orientacji). Użytkownik otrzymuje pas, do którego przymocowanie jest oprzyrządowanie umożliwiające bezprzewodowe podłączenie kasku do komputera.

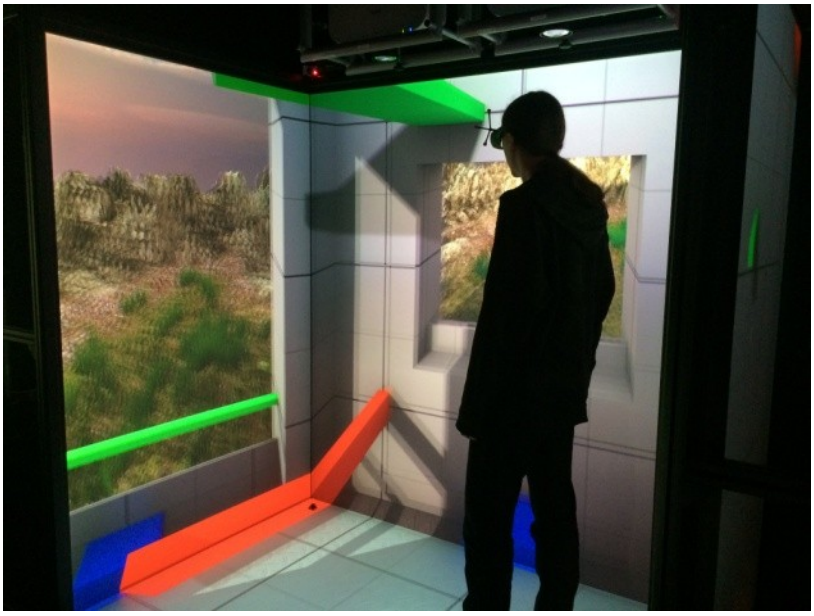
Urządzenie Virtusphere zasilane jest z akumulatora. Podłączane jest złączem USB do komputera, z użyciem bezprzewodowego odbiornika. W systemie operacyjnym widziane jest jako mysz komputerowa.



Rys. 2.6. Sferyczny symulator chodu we wnętrzu BigCAVE'a (po lewej) [21] oraz poza jaskinią, używany z kaskiem rzeczywistości wirtualnej (po prawej) [22]

2.5. MidiCAVE

Średnia jaskinia rzeczywistości wirtualnej – MidiCAVE (rys. 2.7) – ma najczęściej spotykaną formę wśród tego typu instalacji. Składa się bowiem z 4 ekranów: 3 ścian i podłogi. Ekrany ścienne – podobnie jak w BigCAVE'ie – są z akrylu i mają napyloną warstwę projekcyjną. Obraz wyświetlany jest na nich techniką projekcji tylnej. Ekran podłogowy zaś jest niewysokim podestem wyścielonym białą okładziną, na który obraz rzucany jest od góry. Ekran frontowy jest kwadratem o boku długości 2,147 m. Wyświetlany na nim obraz ma rozdzielczość 1920×1920 px i podzielony jest na dwie części – górną i dolną – połączone ze sobą pasem blendingu, stanowiącym ¼ powierzchni ekranu. Ściany boczne mają wysokość 2,147 m, szerokość 1,345 m i rozdzielczość 1200×1920 px. Podłoga ma wymiary 2,147×1,345 m, rozdzielczość 1920×1200 px. Obraz rzucany jest przez 10 projektorów – po dwa na każdy segment: górę frontowego ekranu, jego dół, ścianę lewą, prawą i podłogę.



Rys. 2.7. Średnia jaskinia rzeczywistości wirtualnej – MidiCAVE (fot.: J. Lebiedź)

Zastosowano tu stereoskopię pasywną. Projektory mają założone odpowiednie filtry w takiej konfiguracji, że w każdej parze jeden z nich wyświetla przez cały czas obraz dla lewego oka, a drugi dla prawego. Obraz wyświetlany przez każdy z projektorów ma rozdzielczość 1920×1200 px. Częstotliwość odświeżania obrazu to 60 Hz.

Każda para projektorów obsługiwana jest przez jeden komputer. Wszystkie te węzły zamontowane są w szafie rack, stojącej tuż obok CAVE'a. Każdy z nich jest komputerem Dell Precision Tower 5810, pracującym pod kontrolą systemu Windows 7 (z możliwością przełączenia na system CentOS). Jednostki te wyposażone są w karty graficzne Nvidia Quadro K5200 i procesory Intel Xeon E5-1630 v3, mają po 32 GB pamięci RAM. Przyporządkowano im adresy IP z przedziału 192.168.44.1 – 192.168.44.5 (tabela 2.4). Komputer przypisany górnej części frontowego ekranu jest głównym węzłem tej jaskini.

Tabela 2.4. Przypisanie adresów sieciowych do urządzeń średniej jaskini

Nazwa hosta	**Adres IP**	**Opis**
front-top	192.168.44.1	Komputer dla górnej części ekranu frontowego (główny)
front-bottom	192.168.44.2	Komputer dla dolnej części ekranu frontowego
left	192.168.44.3	Komputer dla ekranu lewego
right	192.168.44.4	Komputer dla ekranu prawego
floor	192.168.44.5	Komputer dla ekranu podłogowego
	192.168.55.101	System śledzenia

W MidiCAVE'ie system śledzenia składa się z jednostki głównej, zamontowanej we wspomnianej wcześniej szafie rack, oraz zestawu 8 kamer zamontowanych w narożnikach jaskini – przy czym „narożniki” tylne są wysunięte do tyłu poza obszar ekranów. Taka liczba i układ kamer pozwala na wykorzystanie w tej jaskini pełnego zestawu śledzenia ciała (*motion capture*) oraz maksymalnie czterech zestawów śledzenia dłoni i palców (*finger tracking*). Poza tym system pozwala na powiązanie z nim do 20 kontrolerów typu Flystick. Śledzić może maksymalnie 50 obiektów (wliczając w to kontrolery i zestawy śledzenia dłoni i palców). Dane z systemu śledzenia przesyłane są do głównego węzła jaskini na dwa porty, z których jeden (7000) jest wykorzystywany przez aplikację ART-Human, służącą wyznaczaniu pozy ciała, a drugi (5000) wykorzystywany może być przez pozostałe aplikacje.

Dźwięk w MidiCAVE'ie dostarczany jest przez zestaw 8 głośników, z których jeden jest głośnikiem superniskotonowym. Źródłem dźwięku jest główny komputer jaskini.

Zgodnie z regulaminem [23] we wnętrzu średniej jaskini mogą znajdować się maksymalnie dwie osoby, natomiast w sali nr 98, w której to stanowisko się znajduje, może przebywać maksymalnie 7 osób.

2.6. MiniCAVE

Mała jaskinia rzeczywistości wirtualnej – MiniCAVE (rys. 2.8) – ma układ ścian analogiczny do jaskini średniej, jednak w tym przypadku są one czterema monitorami ekranowymi. Nie można więc fizycznie wejść do tego CAVE'a, a jedynie „zanurzyć” w nim głowę. Zastosowane ekrany to monitory stereoskopowe wyświetlające obraz w rozdzielczości 2560×1440 px i z odświeżaniem 120 Hz. Ekran ma wymiary 0,6×0,34 m i otoczony jest ramką o szerokości ok. 1 cm. Zastosowano stereoskopię aktywną z okularami migawkowymi.

Rys. 2.8. Mała jaskinia rzeczywistości wirtualnej – MiniCAVE (fot.: J. Lebiedź)

Obraz dla każdego monitora generowany jest przez oddzielny komputer Dell Precision Tower 5810. Jednostki te pracują pod kontrolą systemu Windows 10 i wyposażone są w karty graficzne Nvidia Quadro K5200, procesory Intel Xeon E5-1630 v3 oraz mają po 32 GB pamięci RAM. Komputer

połączony z ekranem frontowym jest głównym węzłem tego CAVE'a. Jest z niego dostęp do sieci internetowej. Przypisanie adresów sieciowych do urządzeń małej jaskini przedstawiono w tabeli 2.5.

Tabela 2.5. Przypisanie adresów sieciowych do urządzeń małej jaskini

Nazwa hosta	**Adres IP**	**Opis**
MINI-FRONT	192.168.66.1	Komputer obsługujący ekran frontowy (główny)
MINI-LEFT	192.168.66.2	Komputer obsługujący ekran lewy
MINI-RIGHT	192.168.66.3	Komputer obsługujący ekran prawy
MINI-BOTTOM	192.168.66.4	Komputer obsługujący ekran dolny
	192.168.77.101	SMARTTRACK – system śledzenia

Dźwięk zapewniony jest przez zestaw 6 głośników, z których jeden jest głośnikiem superniskotonowym. Źródłem dźwięku jest główny komputer jaskini.

MiniCAVE wyposażony jest w system śledzenia SMARTTRACK, który wbudowane ma dwie kamery śledzące. System ten pozwala na powiązanie z nim do 4 kontrolerów typu Flystick. Śledzić może maksymalnie 4 obiekty (wliczając w to kontrolery). Dane z systemu śledzenia przesyłane są na port 5000 do głównego węzła jaskini oraz do stanowisk deweloperskich. Pole widzenia każdej z kamer SMARTTRACKa ma 100° w poziomie oraz 84° w pionie. Maksymalna odległość śledzenia markerów pasywnych to 2,5 m.

2.7. Stanowiska deweloperskie

Aktualnie w LZWP dostępne są 4 stanowiska deweloperskie (rys. 2.9) podłączone do sieci laboratoryjnej. Każde z tych stanowisk wyposażone jest w komputer Dell Precision Tower 5810 z kartą graficzną Nvidia Quadro K5200, procesorem Intel Xeon E5-1630 v3 i 32 GB pamięci RAM. Komputer na stanowisku nr 5 pracuje pod kontrolą systemu Windows 7, pozostałymi zarządza Windows 10. Na każdym stanowisku dostępny jest monitor stereoskopowy, wyświetlający obraz w rozdzielczości 2560×1440 px i z odświeżaniem 120 Hz, oraz para okularów migawkowych dla umożliwienia oglądania obrazu stereoskopowego.

Przypisanie adresów sieciowych do komputerów stanowisk deweloperskich przedstawiono w tabeli 2.6. Wszystkie stanowiska deweloperskie mają dostęp do Internetu.

Żadne z tych stanowisk nie ma własnego systemu śledzenia, jednak korzystać może z systemu SMARTTRACK należącego do MiniCAVE'a. W przypadku stanowisk będących poza zasięgiem widzenia tego systemu śledzenia nie możliwe będzie wykorzystanie go do dopasowania perspektywy dla użytkownika stanowiska deweloperskiego. Powinna natomiast działać komunikacja radiowa, wykorzystywana przez kontrolery Flystick 2.

Tabela 2.6. Przypisanie adresów sieciowych do komputerów stanowisk deweloperskich

Nazwa hosta	Adres IP	Opis
MINI-DEV1	192.168.66.5	Stanowisko deweloperskie nr 1
MINI-DEV2	192.168.66.6	Stanowisko deweloperskie nr 2
MINI-DEV3	192.168.66.7	Stanowisko deweloperskie nr 3
MINI-DEV4	192.168.66.8	Stanowisko deweloperskie nr 4

Rys. 2.9. Jedno ze stanowisk deweloperskich w LZWP

WYKAZ LITERATURY

[1] Unity - Scripting API: Object.DontDestroyOnLoad, https://docs.unity3d.com/ScriptReference/Object.DontDestroyOnLoad.html, (data dostępu 01.09.2020 r.).

[2] Unity - Manual: XR, https://docs.unity3d.com/Manual/XR.html, (data dostępu 01.09.2020 r.).

[3] Unity - Scripting API: XR.XRSettings.LoadDeviceByName, https://docs.unity3d.com/ScriptReference/XR.XRSettings.LoadDeviceByName.html, (data dostępu 01.09.2020 r.).

[4] Unity - Manual: Multi-display, https://docs.unity3d.com/Manual/MultiDisplay.html, (data dostępu 01.09.2020 r.).

[5] Unity - Manual: How to do Stereoscopic Rendering, https://docs.unity3d.com/2017.3/Documentation/Manual/StereoscopicRendering.html, (data dostępu 01.09.2020 r.).

[6] Unity - Manual: Water in Unity, https://docs.unity3d.com/2017.3/Documentation/Manual/HOWTO-Water.html, (data dostępu 01.09.2020 r.).

[7] Unity download archive, https://unity3d.com/get-unity/download/archive, (data dostępu 01.09.2020 r.).

[8] ART GmbH: ART-Human Motion Capture User Manual, v2.1, październik 2014.

[9] Unity - Manual: Network Reference Guide, https://docs.unity3d.com/500/Documentation/Manual/NetworkReferenceGuide.html, (data dostępu 01.09.2020 r.).

[10] What's new in Unity 2018.2, https://unity3d.com/unity/whats-new/unity-2018.2.0#section-backwards-compatibility-breaking-changes, (data dostępu 01.09.2020 r.).

[11] Unity - Manual: RPC Details, https://docs.unity3d.com/500/Documentation/Manual/net-RPCDetails.html, (data dostępu 01.09.2020 r.).

[12] Unity - Manual: Built-in shader variables, https://docs.unity3d.com/Manual/SL-UnityShaderVariables.html, (data dostępu 01.09.2020 r.).

[13] Mixamo, https://www.mixamo.com/ , (data dostępu 01.09.2020 r.).

[14] Nowak M.: Adaptacja algorytmów oświetlenia globalnego do generacji obrazu w czasie rzeczywistym - metoda energetyczna. Praca dyplomowa magisterska. WETI PG, Gdańsk 2016.

[15] Wiszniewski Ł., Ziółkowski T.: Real-Time Connection between Immerse 3D Visualization Laboratory and Kaskada Platform. TASK Quarterly vol. 19, No 4, 2015, s. 471-480.

[16] ART GmbH: System user manual – ARTtrack, TRACKPACK & DTrack, v2.14, lipiec 2018.

[17] Targets, https://ar-tracking.com/products/markers-targets/targets/, (data dostępu 01.09.2020 r.).

[18] Lebiedź J.: Regulamin korzystania z dużej jaskini (sala nr 97) w Laboratorium Zanurzonej Wizualizacji Przestrzennej

[19] Virtusphere, http://virtusphere.com/, (data dostępu 01.09.2020 r.).

[20] Lebiedź J.: Regulamin korzystania ze sferycznego symulatora chodu w Laboratorium Zanurzonej Wizualizacji Przestrzennej

[21] Barco - Gdańsk University of Technology, https://www.barco.com/en/customer-stories/2015/q3/2015-07-03%20-%20gdansk%20university%20of%20technology, (data dostępu 01.09.2020 r.).

[22] Kowalczuk Z., Tatara M.: Sphere Drive and Control System for Haptic Interaction With Physical, Virtual, and Augmented Reality. IEEE Transactions on Control Systems Technology, Vol. PP, Issue 99, 2018, s. 1–15.

[23] Lebiedź J.: Regulamin korzystania ze średniej jaskini (sala nr 98) w Laboratorium Zanurzonej Wizualizacji Przestrzennej

WYKAZ RYSUNKÓW

WYKAZ TABEL

Załącznik nr 1: Przykładowa konfiguracja startowa dla stanowiska BigCAVE

```
{
  "sync": {
    "serverIP": "192.168.11.100",
    "maxConnections": 12
  },
  "input": {
    "dtracks": [
      {
        "name": "DTrack BigCAVE",
        "controllerIP": "192.168.11.101",
        "offsetFromOrigin": [0.0, 1.701, 0.0],
        "bodies": [
          {
            "name": "Glasses Target 4",
            "glasses": true,
            "positionCorrection": [0.0, -0.06, 0.0]
          },
          {
            "name": "Glasses2",
            "glasses": true,
            "positionCorrection": [0.0, -0.06, 0.0]
          }
        ],
        "flysticks": [
          { "name": "Flystick2 Target 1 00428" },
          { "name": "Flystick2 00608" }
        ]
      }
    ]
  },
  "instances": [
    {
      "name": "CAVE-CTRL",
      "cameras": [
        {
          "screenIdx": 0,
          "asymmetricFrustum": false,
          "convergenceDistance": 1.5
        }
      ]
    },
    {
      "name": "FRONT-TOP",
      "cameras": [ { "screenIdx": 1 } ]
    },
    {
      "name": "FRONT-BOTTOM",
      "cameras": [ { "screenIdx": 2 } ]
    },
    {
      "name": "RIGHT-TOP",
      "cameras": [ { "screenIdx": 3 } ]
    },
    {
      "name": "RIGHT-BOTTOM",
      "cameras": [ { "screenIdx": 4 } ]
    },
    {
      "name": "LEFT-TOP",
      "cameras": [ { "screenIdx": 5 } ]
    },
```

```
    {
      "name": "LEFT-BOTTOM",
      "cameras": [ { "screenIdx": 6 } ]
    },
    {
      "name": "BOTTOM-LEFT",
      "cameras": [ { "screenIdx": 7 } ]
    },
    {
      "name": "BOTTOM-RIGHT",
      "cameras": [ { "screenIdx": 8 } ]
    },
    {
      "name": "TOP-LEFT",
      "cameras": [ { "screenIdx": 9 } ]
    },
    {
      "name": "TOP-RIGHT",
      "cameras": [ { "screenIdx": 10 } ]
    },
    {
      "name": "DOOR-TOP",
      "cameras": [ { "screenIdx": 11 } ]
    },
    {
      "name": "DOOR-BOTTOM",
      "cameras": [ { "screenIdx": 12 } ]
    }
  ],
  "pointsOfView": [
    { "poseIdx": 0 },
    { "poseIdx": 1 }
  ],
  "screens": [
    {
      "name": "CAVE-CTRL",
      "centre": [0.0, 0.0, 1.5],
      "width": 3.0,
      "height": 3.0
    },
    {
      "name": "FRONT-TOP",
      "centre": [0.0, 0.638, 1.701],
      "width": 3.402,
      "height": 2.126
    },
    {
      "name": "FRONT-BOTTOM",
      "centre": [0.0, -0.638, 1.701],
      "normal": [0.0, 0.0, -1.0],
      "width": 3.402,
      "height": 2.126
    },


    {
      "name": "RIGHT-TOP",
      "centre": [1.701, 0.638, 0.0],
      "normal": [-1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 2.126
    },
    {
      "name": "RIGHT-BOTTOM",
      "centre": [1.701, -0.638, 0.0],
```

```
      "normal": [-1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 2.126
    },
    {
      "name": "LEFT-TOP",
      "centre": [-1.701, 0.638, 0.0],
      "normal": [1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 2.126
    },
    {
      "name": "LEFT-BOTTOM",
      "centre": [-1.701, -0.638, 0.0],
      "normal": [1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 2.126
    },
    {
      "name": "BOTTOM-LEFT",
      "centre": [-0.638, -1.701, 0.0],
      "normal": [0.0, 1.0, 0.0],
      "up": [-1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 2.126
    },
    {
      "name": "BOTTOM-RIGHT",
      "centre": [0.638, -1.701, 0.0],
      "normal": [0.0, 1.0, 0.0],
      "up": [-1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 2.126
    },
    {
      "name": "TOP-LEFT",
      "centre": [-0.638, 1.701, 0.0],
      "normal": [0.0, -1.0, 0.0],
      "up": [-1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 2.126
    },
    {
      "name": "TOP-RIGHT",
      "centre": [0.638, 1.701, 0.0],
      "normal": [0.0, -1.0, 0.0],
      "up": [-1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 2.126
    },
    {
      "name": "DOOR-TOP",
      "centre": [0.0, 0.638, -1.701],
      "normal": [0.0, 0.0, 1.0],
      "width": 3.402,
      "height": 2.126
    },
    {
      "name": "DOOR-BOTTOM",
      "centre": [0.0, -0.638, -1.701],
      "normal": [0.0, 0.0, 1.0],
      "width": 3.402,
      "height": 2.126
    }
  ],
```

```
"obstacleWalls": [
    {
      "name": "FRONT",
      "centre": [0.0, 0.0, 1.701],
      "width": 3.402,
      "height": 3.402
    },
    {
      "name": "RIGHT",
      "centre": [1.701, 0.0, 0.0],
      "normal": [-1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 3.402
    },
    {
      "name": "LEFT",
      "centre": [-1.701, 0.0, 0.0],
      "normal": [1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 3.402
    },
    {
      "name": "BOTTOM",
      "centre": [0.0, -1.701, 0.0],
      "normal": [0.0, 1.0, 0.0],
      "up": [-1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 3.402
    },
    {
      "name": "TOP",
      "centre": [0.0, 1.701, 0.0],
      "normal": [0.0, -1.0, 0.0],
      "up": [-1.0, 0.0, 0.0],
      "width": 3.402,
      "height": 3.402
    },
    {
      "name": "DOOR",
      "centre": [0.0, 0.0, -1.701],
      "normal": [0.0, 0.0, 1.0],
      "width": 3.402,
      "height": 3.402
    }
  ],
  "screensAndObstacleWallsOffsetFromOrigin": [0.0, 1.701, 0.0],
  "obstacleWallWarningScale": 0.5,
  "custom": {
    "movement": {
      "overwriteSceneMovementType": true,
      "overwriteOtherSceneMovementSettings": true,
      "movementSettings": {
        "general": {
          "movementType": "Flying",
          "dblJumpToToggleMovementType": false
        },
        "flying": {
          "flystick": {
            "moveSpeed": 500.0
          }
        }
      }
    }
  }
}
```

Załącznik nr 2: Przykładowa konfiguracja startowa dla stanowiska MidiCAVE


```
{
  "sync": {
    "serverIP": "192.168.44.1",
    "maxConnections": 4
  },
  "input": {
    "dtracks": [
      {
        "name": "DTrack MidiCAVE",
        "controllerIP": "192.168.55.101",
        "offsetFromOrigin": [0.0, 1.043, 0.0],
        "bodies": [
          {
            "name": "Glasses Target 4 (GT4)",
            "glasses": true,
            "positionCorrection": [0.0, -0.06, 0.0]
          },
          {
            "name": "Glasses Target 5 (GT5)",
            "glasses": true,
            "positionCorrection": [0.0, -0.06, 0.0]
          },
          {
            "name": "Glasses Target 6 (GT6)",
            "glasses": true,
            "positionCorrection": [0.0, -0.06, 0.0]
          },
          { "name": "Lower Leg Target 2 Rev.3.0 (TBT2r3)" },
          { "name": "Hand Target 25 (HT25)" },
          { "name": "Foot Target 1 Rev.3.0 (FT1r3)" },
          { "name": "Forearm Target 1 Rev.3.0 (UBT1r3)" },
          { "name": "Lower Leg Target 1 Rev.3.0 (TBT1r3)" },
          { "name": "Upper Arm Target 2 Rev.3.0 (HBT2r3)" },
          { "name": "Forearm Target 2 Rev.3.0 (UBT2r3)" },
          { "name": "Shoulder Target 1 Rev.3.0 (UT1r3)" },
          { "name": "Foot Target 2 Rev.3.0 (FT2r3)" },
          { "name": "Thigh Target 1 Rev.3.0 (FBT1r3)" },
          { "name": "Upper Arm Target 1 Rev.3.0 (HBT1r3)" },
          { "name": "Hand Target 23 (HT23)" },
          { "name": "Thigh Target 2 Rev.3.0 (FBT2r3)" },
          { "name": "Shoulder Target 2 Rev.3.0 (UT2r3)" },
          { "name": "Back Target 1 Rev_3_0" },
          { "name": "Waist Target 1 Rev_3_0" },
          { "name": "Waist Target 2 Rev_3_0" }
        ],
        "flysticks": [
          { "name": "Flystick2 00614" },
          { "name": "Flystick2 00612" },
          { "name": "Flystick2 00608" },
          { "name": "Flystick2 00613" }
        ],
        "hands": [
          { "name": "Hand Left" },
          { "name": "Hand Right" }
        ]
      }
    ],
```

```
  "arthumans": [
    {
      "name": "ARTHuman MidiCAVE",
      "enabled": true,
      "offsetFromOrigin": [0.0, 1.043, 0.0]
    }
  ]
},
"instances": [
  {
    "name": "FRONT-UP",
    "cameras": [ { "screenIdx": 0 } ]
  },
  {
    "name": "FRONT-BOTTOM",
    "cameras": [ { "screenIdx": 1 } ]
  },
  {
    "name": "BOTTOM",
    "cameras": [ { "screenIdx": 2 } ]
  },
  {
    "name": "LEFT",
    "cameras": [ { "screenIdx": 3 } ]
  },
  {
    "name": "RIGHT",
    "cameras": [ { "screenIdx": 4 } ]
  }
],
"pointsOfView": [
  { "poseIdx": 0 },
  { "poseIdx": 1 }
],
"screens": [
  {
    "name": "FRONT-UP",
    "centre": [0.0, 0.40275, 1.345],
    "width": 2.147,
    "height": 1.3415
  },
  {
    "name": "FRONT-BOTTOM",
    "centre": [0.0, -0.40275, 1.345],
    "width": 2.147,
    "height": 1.3415
  },
  {
    "name": "BOTTOM",
    "centre": [0.0, -1.0735, 0.6725],
    "normal": [0.0, 1.0, 0.0],
    "up": [0.0, 0.0, 1.0],
    "width": 2.147,
    "height": 1.345
  },
  {
    "name": "LEFT",
    "centre": [-1.0735, 0.0, 0.6725],
    "normal": [1.0, 0.0, 0.0],
    "width": 1.345,
    "height": 2.147
  },
  {
    "name": "RIGHT",
    "centre": [1.0735, 0.0, 0.6725],
    "normal": [-1.0, 0.0, 0.0],
```

```
      "width": 1.345,
      "height": 2.147
    }
  ],
  "obstacleWalls": [
    {
      "name": "FRONT",
      "centre": [0.0, 0.0, 1.345],
      "width": 2.147,
      "height": 2.147
    },
    {
      "name": "BOTTOM",
      "centre": [0.0, -1.0735, 0.6725],
      "normal": [0.0, 1.0, 0.0],
      "up": [0.0, 0.0, 1.0],
      "width": 2.147,
      "height": 1.345
    },
    {
      "name": "LEFT",
      "centre": [-1.0735, 0.0, 0.6725],
      "normal": [1.0, 0.0, 0.0],
      "width": 1.345,
      "height": 2.147
    },
    {
      "name": "RIGHT",
      "centre": [1.0735, 0.0, 0.6725],
      "normal": [-1.0, 0.0, 0.0],
      "width": 1.345,
      "height": 2.147
    }
  ],
  "screensAndObstacleWallsOffsetFromOrigin": [0.0, 1.043, 0.0],
  "obstacleWallWarningScale": 0.5,
  "custom": { }
}
```

Załącznik nr 3: Przykładowa konfiguracja startowa dla stanowiska MiniCAVE


```
{
  "sync": {
    "masterInstanceID": 0,
    "serverIP": "192.168.66.1",
    "maxConnections": 10
  },
  "input": {
    "dtracks": [
      {
        "name": "SMARTTRACK miniCave",
        "controllerIP": "192.168.77.101",
        "offsetFromOrigin": [0.0, 1.65, 0.0],
        "bodies": [
          {
            "name": "Glasses Target 8",
            "glasses": true
          },
          {
            "name": "Tree Target 1 (TT1)",
            "glasses": true
          }
        ],
        "flysticks": [
          { "name": "Flystick2 00602" },
          { "name": "Flystick2 00612" }
        ]
      }
    ]
  },
  "instances": [
    {
      "name": "FRONT",
      "cameras": [ { "screenIdx": 0} ]
    },
    {
      "name": "LEFT",
      "cameras": [ { "screenIdx": 1 } ]
    },
    {
      "name": "RIGHT",
      "cameras": [ { "screenIdx": 2 } ]
    },
    {
      "name": "FLOOR",
      "cameras": [ {"screenIdx": 3 } ]
    }
  ],
  "pointsOfView": [
    { "poseIdx": 0 }
  ],
  "screens": [
    {
      "name": "FRONT",
      "centre": [0.0, 0.0, 0.6],
      "width": 0.6,
      "height": 0.34
    },
    {
      "name": "LEFT",
      "centre": [-0.31, 0.0, 0.3],
      "normal": [1.0, 0.0, 0.0],
      "width": 0.6,
```

```
      "height": 0.34
    },
    {
      "name": "RIGHT",
      "centre": [0.31, 0.0, 0.3],
      "normal": [-1.0, 0.0, 0.0],
      "width": 0.6,
      "height": 0.34
    },
    {
      "name": "FLOOR",
      "centre": [0.0, -0.17, 0.42],
      "normal": [0.0, 1.0, 0.0],
      "up": [0.0, 0.0, 1.0],
      "width": 0.6,
      "height": 0.34
    }
  ],
  "obstacleWalls": [
    {
      "name": "FRONT",
      "centre": [0.0, 0.0, 0.6],
      "width": 0.6,
      "height": 0.34
    },
    {
      "name": "LEFT",
      "centre": [-0.31, 0.0, 0.3],
      "normal": [1.0, 0.0, 0.0],
      "width": 0.6,
      "height": 0.34
    },
    {
      "name": "RIGHT",
      "centre": [0.31, 0.0, 0.3],
      "normal": [-1.0, 0.0, 0.0],
      "width": 0.6,
      "height": 0.34
    },
    {
      "name": "FLOOR",
      "centre": [0.0, -0.17, 0.42],
      "normal": [0.0, 1.0, 0.0],
      "up": [0.0, 0.0, 1.0],
      "width": 0.6,
      "height": 0.34
    }
  ],
  "screensAndObstacleWallsOffsetFromOrigin": [0.0, 1.65, 0.0],
  "custom": { }
}
```

Załącznik nr 4: Przykładowa konfiguracja startowa dla stanowiska deweloperskiego


```
{
  "instanceID": null,
  "debug": {
    "logToHtml": true,
    "htmlLogFileName": "log_[INSTANCE_ID]_[DATETIME].html"
  },
  "display": {
    "cursorVisible": true
  },
  "sync": {
    "masterInstanceID": 0,
    "serverIP": "127.0.0.1",
    "port": 54321,
    "sendRate": 60.0,
    "quitOnDisconnection": true,
    "maxConnectionTries": 5,
    "maxConnections": 32
  },
  "input": {
    "posesStartingPosition": [0.0, 1.75, 0.0],
    "posesStartingRotation": [0.0, 0.0, 0.0],
    "dtracks": [
      {
        "name": "SMARTTRACK miniCave",
        "enabled": true,
        "startMeasurement": false,
        "controllerIP": "192.168.77.101",
        "dataPort": 5000,
        "offsetFromOrigin": [0.0, 1.65, 0.0],
        "bodies": [
          {
            "name": "Glasses Target 8",
            "glasses": true,
            "positionCorrection": [0.0, 0.0, 0.0],
            "rotationCorrection": [0.0, 0.0, 0.0]
          },
          {
            "name": "Tree Target 1 (TT1)",
            "glasses": true,
            "positionCorrection": [0.0, 0.0, 0.0],
            "rotationCorrection": [0.0, 0.0, 0.0]
          }
        ],
        "flysticks": [
          {
            "name": "Flystick2 00602",
            "positionCorrection": [0.0, 0.0, 0.0],
            "rotationCorrection": [0.0, 0.0, 0.0]
          },
          {
            "name": "Flystick2 00612",
            "positionCorrection": [0.0, 0.0, 0.0],
            "rotationCorrection": [0.0, 0.0, 0.0]
          }
        ]
      }
    ]
  },
```

```
"instances": [
  {
    "name": "INST_1",
    "cameras": [
      {
        "screenIdx": 0,
        "viewport": [0.0, 0.0, 1.0, 1.0],
        "asymmetricFrustum": false,
        "convergenceDistance": 1.0,
        "pointOfViewIdx": 0
      }
    ]
  },
  {
    "name": "INST_2",
    "cameras": [
      {
        "screenIdx": 0,
        "displayIdx": 0,
        "viewport": [0.0, 0.0, 1.0, 1.0],
        "asymmetricFrustum": false,
        "convergenceDistance": 1.0,
        "pointOfViewIdx": 0
      }
    ]
  }
],
"pointsOfView": [
  { "poseIdx": 0 }
],
"screens": [
  {
    "name": "FRONT",
    "centre": [0.0, 0.0, 0.6],
    "normal": [0.0, 0.0, -1.0],
    "up": [0.0, 1.0, 0.0],
    "width": 0.6,
    "height": 0.34
  }
],
"obstacleWalls": [
  {
    "name": "FRONT",
    "centre": [0.0, 0.0, 0.6],
    "normal": [0.0, 0.0, -1.0],
    "up": [0.0, 1.0, 0.0],
    "width": 0.6,
    "height": 0.34
  }
],
"screensAndObstacleWallsOffsetFromOrigin": [0.0, 1.65, 0.0],
"obstacleWallWarningScale": 0.1,
"setMaxFov": false,
"allowHDR": false,
"allowMSAA": false,
"setRenderingPath": false,
"renderingPath": "DeferredShading",
"custom": {
  "movement": {
    "overwriteSceneMovementType": true,
    "overwriteOtherSceneMovementSettings": true,
    "movementSettings": {
      "general": {
        "movementType": "Flying",
        "dblJumpToToggleMovementType": true,
```

```
        "dblJumpDetectionTime": 0.5,
        "rotationSmoothSpeed": 7.0,
        "keyboardAndMouse": {
          "moveFasterKey": "LeftShift",
          "moveFasterKeyAlt": "None",
          "moveSlowerKey": "LeftControl",
          "moveSlowerKeyAlt": "RightShift",
          "mouseRotationSpeed": 2.0
        },
        "flystick": {
          "anyFlystick": true,
          "flystickIndex": 0,
          "joystickHorizontalAction": "Rotate",
          "rotateSpeed": 1.5
        }
      },
      "flying": {
        "useCapsuleCollider": false,
        "keyboardAndMouse": {
          "flyUpKey": "E",
          "flyUpKeyAlt": "Space",
          "flyDownKey": "Q",
          "NormalMoveSpeed": 7.0,
          "MoveSlowerMultiplier": 0.25,
          "MoveFasterMultiplier": 4.0
        },
        "flystick": {
          "moveSpeed": 10.0
        }
      },
      "walking": {
        "walkingType": "CharacterController",
        "jumpingEnabled": true,
        "flystick": {
          "jumpBtn": "Fire",
          "walkInFlystickDirection": true
        },
        "runSpeedMultiplier": 2.0,
        "strafeSpeedMultiplier": 0.9,
        "backwardSpeedMultiplier": 0.6,
        "stepInterval": 5.0,
        "runstepLenghten": 0.7,
        "playWalkingAudio": true,
        "playJumpingAudio": true,
        "characterControllerWalkingType": {
          "walkSpeed": 5.0,
          "jumpSpeed": 5.0,
          "stickToGroundForce": 10.0,
          "gravityMultiplier": 1.0
        },
        "physicsBasedWalkingType": {
          "walkSpeed": 5.0,
          "JumpForce": 65.0,
          "SlopeCurveModifier": [
            [-90.0, 1.0],
            [0.0, 1.0],
            [90.0, 0.0]
          ],
          "groundCheckDistance": 0.1,
          "stickToGroundHelperDistance": 0.6,
          "airControl": false
        }
      }
    }
  }
}
```