Interpreter opisu działań obiektów – Etap 1.

1 Wprowadzenie

Niniejszy dokument opisuje zakres prac, które należy wykonać w ramach 1. etapu. Etap ten nominalnie trwa od najbliższych zajęć do zajęć następnych. Jednak do najbliższych zajęć należy wykonać wstępne prace, które wymienione są w dalszej części niniejszego opisu.

2 Zakres prac

W ramach pracy nad programem w całym 1. etapie należy zrealizować następujące podzadania:

- Stworzenie czterech wtyczek, które będą ładowane przez program w momencie startu.
- Zaprojektowanie i zdefiniowanie struktur danych, które będą pozwalały łączyć nazwę polecenia, z funkcjami interfejsu wtyczki obsługującej dane polecenie.
- Rozwijanie makr preprocesora w czytanym pliku *programu działań*.
- Wywołanie funkcji właściwej wtyczki dla wczytanego polecenia z pliku działań i wczytanie parametrów działania.
- Wyświetlanie parametrów działania.

Program powinien poprawnie wczytywać uproszczoną sekwencję poleceń bez słów `Begin_Parallel_Actions` i `End_Parallel_Actions`.

3 Zakres prac wstępnych

Wymienione poniżej podzadania należy zrealizować do najbliższych zajęć. Stanowią one część zadań wymienionych we wcześniejszym rozdziale. Wspomniane podzadania to:

- Bazując na dostarczonym zalążku należy stworzyć drugą *wtyczkę* i ją poprawnie skompilować oraz skonsolidować. Podobnie jak w przypadku pierwszej wtyczki, należy ją załadować w funkcji `main` oraz uruchomić odpowiednie elementy jej interfejsu, tak jak to ma miejsce w przypadku wspomnianej pierwszej *wtyczki*.
- Rozwijanie makr preprocesora w czytanym pliku *programu działań* i wyświetlenie zawartości pliku po przetworzeniu go przez preprocesor języka C.

4 Struktury danych

Proponuje się, aby w programie była zdefiniowana klasa `Scene`. `MobileObj`. Zestaw niezbędnych metod przedstawiony jest zalążkach definicji tej klasy przedstawionym w dalszej części.

Do obowiązkowych klas należą:

- klas abstrakcyjna `Interp4Command` , która definiuje interfejs wtyczki i jest klasą bazową dla klas interpreterów poleceń definiowanej dla każdej z wtyczki,
- klasa `MobileObj`, która jest klasą bazową dla typów obiektów mogących pojawić się na scenie.

4.1 Scena

Propozycja klasy opcjonalnej:

```
class Scena {
      ...
   public:
      ...
    MobileObj*  FindMobileObj(const char *sName);
    void AddMobileObj(MobileObj *pMobObj);
};
```

4.2 Interp4Command

Klasa obowiązkowa abstrakcyjna definiująca interfejs wtyczek. Jest ona klasą bazową dla klas implementujących interpretery poleceń w poszczególnych wtyczkach.

```
class Interp4Command {
 public:
  /*!
   * \brief Wyświetla postać bieżącego polecenia (nazwę oraz wartości parametrów)
   */
   virtual void PrintCmd() const = 0;
  /*!
   * \brief Wyświetla składnię polecenia
   */
   virtual void PrintSyntax() const = 0;
   /*!
    * \brief Wyświetla nazwę polecenia
    */
   virtual const char* GetCmdName() const = 0;
   /*!
    * \brief Wykonuje polecenie oraz wizualizuje jego realizację
    */
   virtual bool ExecCmd(MobileObj *wObMob, int Socket) = 0;
   /*!
    * \brief Czyta wartości parametrów danego polecenia
    */
   virtual bool ReadParams(std::istream& Strm_CmdsList) = 0;
};
```

4.3 MobileObj

Klasa obowiązkowa stanowiąca składnik interfejsu wtyczek. Jest ona klasą abstrakcyjną definiującą interfejs do typów obiektów mobilnych, które mogą pojawić się na scenie. Moduł klasy `Vector3D` jest dostarczony w nowej wersji zalążka do zadania.

```
/*!
 * Nazwy pól klasy są jedynie propozycją i mogą być zmienione
```

```
* Nazwy metod są obowiązujące.
*/
class MobileObj {
   /*!
    * \brief Kąt \e yaw reprezentuje rotację zgodnie z ruchem wskazówek zegara
    *        wokół osi \e OZ.
    */
   double _Ang_Yaw_deg = 0;

   /*!
    * \brief Kąt \e pitch reprezentuje rotację zgodnie z ruchem wskazówek zegara
    *        wokół osi \e OY.
    */
   double _Ang_Pitch_deg = 0;

   /*!
    * \brief Kąt \e roll reprezentuje rotację zgodnie z ruchem wskazówek zegara
    *        wokół osi \e OX.
    */
   double _Ang_Roll_deg = 0;

   /*!
    * \brief Współrzędne aktualnej pozycji obiektu
    *
    * Współrzędne aktualnej pozycji obiektu. Przyjmuje się,
    * że współrzędne wyrażone są w metrach.
    */
   Vector3D  _Position_m;

   /*!
    * \brief Nazwa obiektu, która go indentyfikuje.
    *
    * Nazwa obiektu, która go indentyfikuje. Z tego względu musi
    * musi to być nazwa unikalna wśród wszystkich obiektów na scenie.
    */
   std::string  _Name;

 public:

   double GetAng_Roll_deg() const { return _Ang_Roll_deg; }
   double GetAng_Pitch_deg() const { return _Ang_Pitch_deg; }
   double GetAng_Yaw_deg() const { return _Ang_Yaw_deg; }

   void SetAng_Roll_deg(double Ang_Roll_deg) { _Ang_Roll_deg = Ang_Roll_deg; }
   void SetAng_Pitch_deg(double Ang_Pitch_deg) { _Ang_Pitch_deg = Ang_Pitch_deg; }
   void SetAng_Yaw_deg(double Ang_Yaw_deg) { _Ang_Yaw_deg = Ang_Yaw_deg; }

   const Vector3D & GetPositoin_m() const { return _Position_m; }
   void SetPosition_m(const Vector3D &rPos) { _Position = rPos; }
   Vector3D & UsePosition_m() { return _Position_m; }

  /*!
   * \brief Zmienia nazwę obiektu.
   */
   void SetName(const char* sName) { _Name = sName; }
   /*!
    * \brief Udostępnia nazwę obiektu.
```

```
     */
    const std::string & GetName() const { return _Name; }
};
```

5 Interfejs wtyczek

Każda wtyczka realizująca pojedynczą komendę powinna udostępniać następujące funkcje:

`const char* GetCmdName()` – zwraca wskaźnik do napisu będącego nazwą danego polecenia. W przypadku polecenia `Move` będzie to wskaźnik na napis ''`Move`''.

`Interp4Command* CreateCmd()` – tworzy obiekt modelujący dane polecenie.