```cpp
#define UNICODE

#include "KinovaTypes.h"
#include <Windows.h>
#include "CommunicationLayerWindows.h"
#include "CommandLayer.h"
#include <conio.h>
#include <SFML/Graphics.hpp>
#include <astra/astra.hpp>
#include <cstring>
#include <iostream>
#include <thread>
#include <atomic>

using namespace std;

HINSTANCE  commandLayer_handle;

//Function pointers to the functions we need
int(*MyInitAPI)();
int(*MyCloseAPI)();
int(*MySendBasicTrajectory)(TrajectoryPoint command);
int(*MyGetDevices)(KinovaDevice devices[MAX_KINOVA_DEVICE], int &result);
int(*MySetActiveDevice)(KinovaDevice device);
int(*MyMoveHome)();
int(*MyInitFingers)();
int(*MyGetCartesianCommand)(CartesianPosition &);


astra::Vector3f Right_Hand_Pos = astra::Vector3f();
astra::Vector3f Left_Hand_Pos = astra::Vector3f();


//global variables
int NumofBodies = 0;
int FirstDetect = 0;
int Right_Hand_Grip = -1;
int Left_Hand_Grip = -1;
int Bodyflag = 0;
double rob_pos[3];
double Dtogoal = 1000;
double GUrep_bnd[] = { 0,0,0 };
double GUrep_obs[] = { 0,0,0 };
double GUrep[] = { 0,0,0 };
double GUatt[3];
double gradient[3];
double D;
double DtoCenter;
double Cons;
double norm_gradient;
int numloop = 0;
double bnd[2][3] = { { -0.3,-0.6,-0.2 },{ 0.5,0.0,0.6 } };  //boundary
double bnd_center[] = { 0.5*(bnd[1][1] + bnd[2][1]),0.5*(bnd[1][2] + bnd[2]
  [2]),0.5*(bnd[1][3] + bnd[2][3]) };
int T_gap = 1200;
int c_gap = -5000;
```

```
56  double norm_momentum = 0.0;
57
58  #define PI 3.141592
59
60  class sfLine : public sf::Drawable
61  {
62  public:
63      sfLine(const sf::Vector2f& point1, const sf::Vector2f& point2, sf::Color
        color, float thickness)
64          : color_(color)
65      {
66          const sf::Vector2f direction = point2 - point1;
67          const sf::Vector2f unitDirection = direction / std::sqrt
            (direction.x*direction.x + direction.y*direction.y);
68          const sf::Vector2f normal(-unitDirection.y, unitDirection.x);
69
70          const sf::Vector2f offset = (thickness / 2.f) * normal;
71
72          vertices_[0].position = point1 + offset;
73          vertices_[1].position = point2 + offset;
74          vertices_[2].position = point2 - offset;
75          vertices_[3].position = point1 - offset;
76
77          for (int i = 0; i<4; ++i)
78              vertices_[i].color = color;
79      }
80
81      void draw(sf::RenderTarget &target, sf::RenderStates states) const
82      {
83          target.draw(vertices_, 4, sf::Quads, states);
84      }
85
86  private:
87      sf::Vertex vertices_[4];
88      sf::Color color_;
89  };
90
91  class BodyVisualizer : public astra::FrameListener
92  {
93  public:
94      static sf::Color get_body_color(std::uint8_t bodyId)
95      {
96          if (bodyId == 0)
97          {
98              // Handle no body separately - transparent
99              return sf::Color(0x00, 0x00, 0x00, 0x00);
100         }
101         // Case 0 below could mean bodyId == 25 or
102         // above due to the "% 24".
103         switch (bodyId % 6) {
104         case 0:
105             return sf::Color(0x00, 0x88, 0x00, 0xFF);
106         case 1:
107             return sf::Color(0x00, 0x00, 0xFF, 0xFF);
108         case 2:
109             return sf::Color(0x88, 0x00, 0x00, 0xFF);
```

```
110         case 3:
111             return sf::Color(0x00, 0xFF, 0x00, 0xFF);
112         case 4:
113             return sf::Color(0x00, 0x00, 0x88, 0xFF);
114         case 5:
115             return sf::Color(0xFF, 0x00, 0x00, 0xFF);
116         default:
117             return sf::Color(0xAA, 0xAA, 0xAA, 0xFF);
118         }
119     }
120
121     void init_depth_texture(int width, int height)
122     {
123         if (displayBuffer_ == nullptr || width != depthWidth_ || height !=
              depthHeight_)
124         {
125             depthWidth_ = width;
126             depthHeight_ = height;
127             int byteLength = depthWidth_ * depthHeight_ * 4;
128
129             displayBuffer_ = BufferPtr(new uint8_t[byteLength]);
130             std::memset(displayBuffer_.get(), 0, byteLength);
131
132             texture_.create(depthWidth_, depthHeight_);
133             sprite_.setTexture(texture_, true);
134             sprite_.setPosition(0, 0);
135         }
136     }
137
138     void init_overlay_texture(int width, int height)
139     {
140         if (overlayBuffer_ == nullptr || width != overlayWidth_ || height !=
              overlayHeight_)
141         {
142             overlayWidth_ = width;
143             overlayHeight_ = height;
144             int byteLength = overlayWidth_ * overlayHeight_ * 4;
145
146             overlayBuffer_ = BufferPtr(new uint8_t[byteLength]);
147             std::fill(&overlayBuffer_[0], &overlayBuffer_[0] + byteLength, 0);
148
149             overlayTexture_.create(overlayWidth_, overlayHeight_);
150             overlaySprite_.setTexture(overlayTexture_, true);
151             overlaySprite_.setPosition(0, 0);
152         }
153     }
154
155     void check_fps()
156     {
157         double fpsFactor = 0.02;
158
159         std::clock_t newTimepoint = std::clock();
160         long double frameDuration = (newTimepoint - lastTimepoint_) /
              static_cast<long double>(CLOCKS_PER_SEC);
161
162         frameDuration_ = frameDuration * fpsFactor + frameDuration_ * (1 -
```

```
            fpsFactor);
163         lastTimepoint_ = newTimepoint;
164         double fps = 1.0 / frameDuration_;
165
166         //printf("FPS: %3.1f (%3.4Lf ms)\n", fps, frameDuration_ * 1000);
167     }
168
169     void processDepth(astra::Frame& frame)
170     {
171         const astra::DepthFrame depthFrame = frame.get<astra::DepthFrame>();
172
173         if (!depthFrame.is_valid()) { return; }
174
175         int width = depthFrame.width();
176         int height = depthFrame.height();
177
178         init_depth_texture(width, height);
179
180         const int16_t* depthPtr = depthFrame.data();
181         for (int y = 0; y < height; y++)
182         {
183             for (int x = 0; x < width; x++)
184             {
185                 int index = (x + y * width);
186                 int index4 = index * 4;
187
188                 int16_t depth = depthPtr[index];
189                 uint8_t value = depth % 255;
190                 // Normalize depth
191                 // uint8_t value = round((depth / 3400) * 255);         // Too
                    Dark..
192
193                 displayBuffer_[index4] = value;
194                 displayBuffer_[index4 + 1] = value;
195                 displayBuffer_[index4 + 2] = value;
196                 displayBuffer_[index4 + 3] = 255;
197             }
198         }
199
200         texture_.update(displayBuffer_.get());
201     }
202
203     void processBodies(astra::Frame& frame)
204     {
205         astra::BodyFrame bodyFrame = frame.get<astra::BodyFrame>();
206
207         jointPositions_.clear();
208         circles_.clear();
209         circleShadows_.clear();
210         boneLines_.clear();
211         boneShadows_.clear();
212
213         if (!bodyFrame.is_valid() || bodyFrame.info().width() == 0 ||
              bodyFrame.info().height() == 0)
214         {
215             clear_overlay();
```

```
216             NumofBodies = 0;
217             return;
218         }
219
220         const float jointScale = bodyFrame.info().width() / 120.f;
221
222         const auto& bodies = bodyFrame.bodies();
223
224         // Detect Human Body -> Immediately Stop
225         NumofBodies = bodies.size();
226         if (NumofBodies > 0 && FirstDetect == 0) {
227             FirstDetect = 1;
228             return;
229         }
230
231         for (auto& body : bodies)
232         {
233             /*printf("Processing frame #%d body %d left hand: %u\n",
234             bodyFrame.frame_index(), body.id(), unsigned(body.hand_poses
                ().left_hand()))*/;
235         for (auto& joint : body.joints())
236         {
237             jointPositions_.push_back(joint.depth_position());
238         }
239
240         update_body(body, jointScale);
241         }
242
243         const auto& floor = bodyFrame.floor_info(); //floor
244         if (floor.floor_detected())
245         {
246             const auto& p = floor.floor_plane();
247
248         }
249
250         const auto& bodyMask = bodyFrame.body_mask();
251         const auto& floorMask = floor.floor_mask();
252
253         update_overlay(bodyMask, floorMask);
254     }
255
256     void update_body(astra::Body body,
257         const float jointScale)
258     {
259         const auto& joints = body.joints();
260
261         if (joints.empty())
262         {
263             return;
264         }
265
266         for (const auto& joint : joints)
267         {
268             astra::JointType type = joint.type();
269             const auto& pos = joint.depth_position();
270
```

```
271             if (joint.status() == astra::JointStatus::NotTracked)
272             {
273                 continue;
274             }
275
276             auto radius = jointRadius_ * jointScale; // pixels
277             sf::Color circleShadowColor(0, 0, 0, 255);
278
279             auto color = sf::Color(0x00, 0xFF, 0x00, 0xFF);
280
281             if (type == astra::JointType::LeftHand)
282             {
283                 if (astra::HandPose::Grip == body.hand_poses().left_hand()) {
284                     Left_Hand_Grip = 1;
285                     radius *= 1.5f;
286                     circleShadowColor = sf::Color(255, 255, 255, 255);
287                     color = sf::Color(0x00, 0xAA, 0xFF, 0xFF);
288                 }
289                 else {
290                     Left_Hand_Grip = 0;
291                 }
292             }
293
294             if (type == astra::JointType::RightHand)
295             {
296                 if (astra::HandPose::Grip == body.hand_poses().right_hand()) {
297                     Right_Hand_Grip = 1;
298                     radius *= 1.5f;
299                     circleShadowColor = sf::Color(255, 255, 255, 255);
300                     color = sf::Color(0x00, 0xAA, 0xFF, 0xFF);
301                 }
302                 else {
303                     Right_Hand_Grip = 0;
304                 }
305             }
306
307             const auto shadowRadius = radius + shadowRadius_ * jointScale;
308             const auto radiusDelta = shadowRadius - radius;
309
310             sf::CircleShape circle(radius);
311
312             circle.setFillColor(sf::Color(color.r, color.g, color.b, 255));
313             circle.setPosition(pos.x - radius, pos.y - radius);
314             circles_.push_back(circle);
315
316             sf::CircleShape shadow(shadowRadius);
317             shadow.setFillColor(circleShadowColor);
318             shadow.setPosition(circle.getPosition() - sf::Vector2f
                  (radiusDelta, radiusDelta));
319             circleShadows_.push_back(shadow);
320         }
321
322         update_bone(joints, jointScale, astra::JointType::Head,
              astra::JointType::ShoulderSpine);
323
324         update_bone(joints, jointScale, astra::JointType::ShoulderSpine,
```

```
            astra::JointType::LeftShoulder);
325         update_bone(joints, jointScale, astra::JointType::LeftShoulder,
              astra::JointType::LeftElbow);
326         update_bone(joints, jointScale, astra::JointType::LeftElbow,
              astra::JointType::LeftHand);
327
328         update_bone(joints, jointScale, astra::JointType::ShoulderSpine,
              astra::JointType::RightShoulder);
329         update_bone(joints, jointScale, astra::JointType::RightShoulder,
              astra::JointType::RightElbow);
330         update_bone(joints, jointScale, astra::JointType::RightElbow,
              astra::JointType::RightHand);
331
332         update_bone(joints, jointScale, astra::JointType::ShoulderSpine,
              astra::JointType::MidSpine);
333         update_bone(joints, jointScale, astra::JointType::MidSpine,
              astra::JointType::BaseSpine);
334
335         update_bone(joints, jointScale, astra::JointType::BaseSpine,
              astra::JointType::LeftHip);
336         update_bone(joints, jointScale, astra::JointType::LeftHip,
              astra::JointType::LeftKnee);
337         update_bone(joints, jointScale, astra::JointType::LeftKnee,
              astra::JointType::LeftFoot);
338
339         update_bone(joints, jointScale, astra::JointType::BaseSpine,
              astra::JointType::RightHip);
340         update_bone(joints, jointScale, astra::JointType::RightHip,
              astra::JointType::RightKnee);
341         update_bone(joints, jointScale, astra::JointType::RightKnee,
              astra::JointType::RightFoot);
342     }
343
344     void update_bone(const astra::JointList& joints,
345         const float jointScale, astra::JointType j1,
346         astra::JointType j2)
347     {
348         const auto& joint1 = joints[int(j1)];
349         const auto& joint2 = joints[int(j2)];
350         const auto& jp1w = joint1.world_position();
351         const auto& jp2w = joint2.world_position();
352
353         switch (j1) {
354         case astra::JointType::LeftElbow:
355             switch (j2) {
356             case astra::JointType::LeftHand:
357                 Left_Hand_Pos = jp2w;
358             }
359         case astra::JointType::RightElbow:
360             switch (j2) {
361             case astra::JointType::RightHand:
362                 Right_Hand_Pos = jp2w;
363             }
364         }
365
366         if (joint1.status() == astra::JointStatus::NotTracked ||
```

```
367             joint2.status() == astra::JointStatus::NotTracked)
368         {
369             //don't render bones between untracked joints
370             return;
371         }
372
373         const auto& jp1 = joint1.depth_position();
374         const auto& jp2 = joint2.depth_position();
375
376         auto p1 = sf::Vector2f(jp1.x, jp1.y);
377         auto p2 = sf::Vector2f(jp2.x, jp2.y);
378
379         sf::Color color(255, 255, 255, 255);
380         float thickness = lineThickness_ * jointScale;
381         if (joint1.status() == astra::JointStatus::LowConfidence ||
382             joint2.status() == astra::JointStatus::LowConfidence)
383         {
384             color = sf::Color(128, 128, 128, 255);
385             thickness *= 0.5f;
386         }
387
388         boneLines_.push_back(sfLine(p1,
389             p2,
390             color,
391             thickness));
392         const float shadowLineThickness = thickness + shadowRadius_ *
              jointScale * 2.f;
393         boneShadows_.push_back(sfLine(p1,
394             p2,
395             sf::Color(0, 0, 0, 255),
396             shadowLineThickness));
397     }
398
399     void update_overlay(const astra::BodyMask& bodyMask,
400         const astra::FloorMask& floorMask)
401     {
402         const auto* bodyData = bodyMask.data();
403         const auto* floorData = floorMask.data();
404         const int width = bodyMask.width();
405         const int height = bodyMask.height();
406
407         init_overlay_texture(width, height);
408
409         const int length = width * height;
410
411         for (int i = 0; i < length; i++)
412         {
413             const auto bodyId = bodyData[i];
414             const auto isFloor = floorData[i];
415
416             sf::Color color(0x0, 0x0, 0x0, 0x0);
417
418             if (bodyId != 0)
419             {
420                 color = get_body_color(bodyId);
421             }
```

```
422             else if (isFloor != 0)
423             {
424                 color = sf::Color(0x0, 0x0, 0xFF, 0x88);
425             }
426 
427             const int rgbaOffset = i * 4;
428             overlayBuffer_[rgbaOffset] = color.r;
429             overlayBuffer_[rgbaOffset + 1] = color.g;
430             overlayBuffer_[rgbaOffset + 2] = color.b;
431             overlayBuffer_[rgbaOffset + 3] = color.a;
432         }
433 
434         overlayTexture_.update(overlayBuffer_.get());
435     }
436 
437     void clear_overlay()
438     {
439         int byteLength = overlayWidth_ * overlayHeight_ * 4;
440         std::fill(&overlayBuffer_[0], &overlayBuffer_[0] + byteLength, 0);
441 
442         overlayTexture_.update(overlayBuffer_.get());
443     }
444 
445     virtual void on_frame_ready(astra::StreamReader& reader,
446         astra::Frame& frame) override
447     {
448         processDepth(frame);
449         processBodies(frame);
450 
451         check_fps();
452     }
453 
454     void draw_bodies(sf::RenderWindow& window)
455     {
456         const float scaleX = window.getView().getSize().x / overlayWidth_;
457         const float scaleY = window.getView().getSize().y / overlayHeight_;
458 
459         sf::RenderStates states;
460         sf::Transform transform;
461         transform.scale(scaleX, scaleY);
462         states.transform *= transform;
463 
464         for (const auto& bone : boneShadows_)
465             window.draw(bone, states);
466 
467         for (const auto& c : circleShadows_)
468             window.draw(c, states);
469 
470         for (const auto& bone : boneLines_)
471             window.draw(bone, states);
472 
473         for (auto& c : circles_)
474             window.draw(c, states);
475 
476     }
477 
```

```
478     void draw_to(sf::RenderWindow& window)
479     {
480         if (displayBuffer_ != nullptr)
481         {
482             const float scaleX = window.getView().getSize().x / depthWidth_;
483             const float scaleY = window.getView().getSize().y / depthHeight_;
484             sprite_.setScale(scaleX, scaleY);
485 
486             window.draw(sprite_); // depth
487         }
488 
489         if (overlayBuffer_ != nullptr)
490         {
491             const float scaleX = window.getView().getSize().x / overlayWidth_;
492             const float scaleY = window.getView().getSize().y /
                  overlayHeight_;
493             overlaySprite_.setScale(scaleX, scaleY);
494             window.draw(overlaySprite_); //bodymask and floormask
495         }
496 
497         draw_bodies(window);
498     }
499 
500 private:
501     long double frameDuration_{ 0 };
502     std::clock_t lastTimepoint_{ 0 };
503     sf::Texture texture_;
504     sf::Sprite sprite_;
505 
506     using BufferPtr = std::unique_ptr < uint8_t[] >;
507     BufferPtr displayBuffer_{ nullptr };
508 
509     std::vector<astra::Vector2f> jointPositions_;
510 
511     int depthWidth_{ 0 };
512     int depthHeight_{ 0 };
513     int overlayWidth_{ 0 };
514     int overlayHeight_{ 0 };
515 
516     std::vector<sfLine> boneLines_;
517     std::vector<sfLine> boneShadows_;
518     std::vector<sf::CircleShape> circles_;
519     std::vector<sf::CircleShape> circleShadows_;
520 
521     float lineThickness_{ 0.5f }; // pixels
522     float jointRadius_{ 1.0f };   // pixels
523     float shadowRadius_{ 0.5f };  // pixels
524 
525     BufferPtr overlayBuffer_{ nullptr };
526     sf::Texture overlayTexture_;
527     sf::Sprite overlaySprite_;
528 
529 };
530 
531 astra::DepthStream configure_depth(astra::StreamReader& reader)
532 {
```

```
533     auto depthStream = reader.stream<astra::DepthStream>();
534
535     //We don't have to set the mode to start the stream, but if you want to
          here is how:
536     astra::ImageStreamMode depthMode;
537
538     depthMode.set_width(640);
539     depthMode.set_height(480);
540     depthMode.set_pixel_format
          (astra_pixel_formats::ASTRA_PIXEL_FORMAT_DEPTH_MM);
541     depthMode.set_fps(30);
542
543     depthStream.set_mode(depthMode);
544
545     return depthStream;
546 }
547
548 void thread_hand(atomic<bool>& flag, float* xgoal, float* ygoal, float* zgoal)

549 {
550
551     // Astra Camera Intialization [Start]
552     std::cout << "Start Camera Initialization" << endl;
553     astra::initialize();
554     const char* licenseString = "<INSERT LICENSE KEY HERE>";
555     orbbec_body_tracking_set_license(licenseString);
556
557     sf::RenderWindow window(sf::VideoMode(1280, 960), "Simple Body Viewer");
558
559
560     auto fullscreenStyle = sf::Style::None;
561     const sf::VideoMode fullScreenMode = sf::VideoMode::getFullscreenModes()
          [0];
562     const sf::VideoMode windowedMode(1280, 960);
563     bool isFullScreen = false;
564
565     astra::StreamSet sensor;
566     astra::StreamReader reader = sensor.create_reader();
567
568     BodyVisualizer listener;
569
570     auto depthStream = configure_depth(reader);
571     depthStream.start();
572
573     auto bodyStream = reader.stream<astra::BodyStream>();
574     bodyStream.start();
575     reader.add_listener(listener);
576
577     astra::SkeletonProfile profile = bodyStream.get_skeleton_profile();
578
579     // HandPoses includes Joints and Segmentation
580     astra::BodyTrackingFeatureFlags features =
          astra::BodyTrackingFeatureFlags::HandPoses;
581
582     // Astra Camera Intialization [End]
583
```

```
584
585     while (flag)
586     {
587             astra_update();
588
589             if (NumofBodies > 0)
590             {
591                 if (Bodyflag == 0)
592                 {
593                     Bodyflag = 1;
594                 }
595                 else //Bodyflag == 1
596                 {
597                     //hand position in Kinova coordinate
598                     float L_X = ((Left_Hand_Pos.x * 0.001) - 0.22);
599                     float L_Y = (-sin(PI / 4.0f)*(Left_Hand_Pos.z * 0.001) -
                    sin(PI / 4.0f)*(Left_Hand_Pos.y * 0.001) - 0.05);
600                     float L_Z = (-sin(PI / 4.0f)*(Left_Hand_Pos.z * 0.001) +
                    sin(PI / 4.0f)*(Left_Hand_Pos.y * 0.001) + 1.15);
601                     float R_X = ((Right_Hand_Pos.x * 0.001) - 0.22);
602                     float R_Y = (-sin(PI / 4.0f)*(Right_Hand_Pos.z * 0.001) -
                    sin(PI / 4.0f)*(Right_Hand_Pos.y * 0.001) - 0.05);
603                     float R_Z = (-sin(PI / 4.0f)*(Right_Hand_Pos.z * 0.001) +
                    sin(PI / 4.0f)*(Right_Hand_Pos.y * 0.001) + 1.15);
604
605                     //update goal position (hand pos)
606                     *xgoal = R_X + 0.0f;
607                     *ygoal = R_Y + 0.6f;
608                     *zgoal = R_Z + 0.0f;
609                 }
610             }
611
612             window.clear(sf::Color::Black);
613             listener.draw_to(window);
614             window.display();
615     }
616
617
618     return;
619 }
620
621 int main()
622 {
623     //test case 1 - hand tracking w/o momentum
624     //double Kappa = 0.4;  // Attractive Potential Gain
625     //double Nu = 1.0e-6;  // Repulsive Potential Gain
626     //double ObsTh = 0.05; // Obstacle
627     //double ObsTh = 0.03;
628     //double start[] = { 0.1,-0.3,0.5 };
629     //double start_theta[] = { -3.14,0.0,0.0 };
630     //double goal[] = { 0.0,0.0,0.0 };
631     //double obs[2][4] = { { 0.15,-0.3,0.28,0.05 },{ 0.2,-0.5,0.22,0.04 } };
632     //double stepsize = 0.01;
633     //int obsnum = 0;
634
635     //test case 2 - w/ momentum, 1 ball
```

```
636     //double Kappa = 0.4;
637     //double Nu = 1.0e-6;
638     //double rate = 0.9;
639     //double ObsTh = 0.05;
640     //double start[] = { 0.034,-0.2,0.26 };
641     //double temp[] = { 0,0,0 };
642     //double start_theta[] = { -3.14,0.0,0.0 };
643     //double goal[] = { 0.27,-0.60,-0.02 };
644     //double momentum[] = { 0,0,0 };
645     //double obs[1][4] = { { 0.18,-0.45,0.08,0.13 } };
646     //double stepsize = 0.01;
647     //int obsnum = 1;
648     //double goal_theta[] = { 3.14,0.0,0.0 };
649
650     //test case 3 - w/ momentum, 2 balls
651     double Kappa = 0.4;
652     double Nu = 1.0e-6;
653     double rate = 0.9;
654     double ObsTh = 0.05;
655     double start[] = { 0.034,-0.2,0.26 };
656     double temp[] = { 0,0,0 };
657     double start_theta[] = { -3.14,0.0,0.0 };
658     double goal[] = { 0.0,0.0,0.0 };
659     double momentum[] = { 0,0,0 };
660     double obs[2][4] = { { 0.237,-0.29,0.08,0.12 },
          { 0.085,-0.49,0.02,0.12 } };
661     double stepsize = 0.01;
662     int obsnum = 2;
663     double goal_theta[] = { 3.14,0.0,0.0 };
664
665
666     int programResult = 0;
667
668     commandLayer_handle = LoadLibrary(L"CommandLayerWindows.dll");
669
670     //We load the functions from the library
671     MyInitAPI = (int(*)()) GetProcAddress(commandLayer_handle, "InitAPI");
672     MyCloseAPI = (int(*)()) GetProcAddress(commandLayer_handle, "CloseAPI");
673     MyMoveHome = (int(*)()) GetProcAddress(commandLayer_handle, "MoveHome");
674     MyInitFingers = (int(*)()) GetProcAddress(commandLayer_handle,
          "InitFingers");
675     MyGetDevices = (int(*)(KinovaDevice devices[MAX_KINOVA_DEVICE], int
          &result)) GetProcAddress(commandLayer_handle, "GetDevices");
676     MySetActiveDevice = (int(*)(KinovaDevice devices)) GetProcAddress
          (commandLayer_handle, "SetActiveDevice");
677     MySendBasicTrajectory = (int(*)(TrajectoryPoint)) GetProcAddress
          (commandLayer_handle, "SendBasicTrajectory");
678     MyGetCartesianCommand = (int(*)(CartesianPosition &)) GetProcAddress
          (commandLayer_handle, "GetCartesianCommand");
679
680     //Verify that all functions has been loaded correctly
681     if ((MyInitAPI == NULL) || (MyCloseAPI == NULL) || (MySendBasicTrajectory
          == NULL) ||
682         (MyGetDevices == NULL) || (MySetActiveDevice == NULL) ||
              (MyGetCartesianCommand == NULL) ||
683         (MyMoveHome == NULL) || (MyInitFingers == NULL))
```

```
684
685     {
686         std::cout << "* * *  E R R O R  D U R I N G  I N I T I A L I Z A T I O N  * * *" << endl;
687         programResult = 0;
688         return 0;
689     }
690     else
691     {
692         std::cout << "I N I T I A L I Z A T I O N  C O M P L E T E D - M A I N" << endl << endl;
693     }
694     int result = (*MyInitAPI)();
695
696     std::cout << "Main Initialization's result :" << result << endl;
697
698     KinovaDevice list[MAX_KINOVA_DEVICE];
699
700     int devicesCount = MyGetDevices(list, result);
701
702     std::cout << "Found a robot on the USB bus (" << list[0].SerialNumber << ")" << endl;
703
704     //Setting the current device as the active device.
705     MySetActiveDevice(list[0]);
706
707     std::cout << "Send the robot to Home position" << endl;
708     MyMoveHome();
709
710     std::cout << "Initializing the fingers" << endl;
711     MyInitFingers();
712
713     TrajectoryPoint pointToSend;
714     pointToSend.InitStruct();
715     pointToSend.Position.Type = CARTESIAN_POSITION;
716
717     atomic<bool> flag = true ;
718
719     float xp, yp, zp = 0;
720     float* xgoal = &xp;
721     float* ygoal = &yp;
722     float* zgoal = &zp;
723
724     std::cout << "********************START**************************" << endl;
725
726     std::thread hand_t(&thread_hand, ref(flag), xgoal, ygoal, zgoal);
727
728     CartesianPosition currentPosition;
729
730     //Sending to start position
731     MyGetCartesianCommand(currentPosition);
732     pointToSend.Position.CartesianPosition.Z = start[2];
733     pointToSend.Position.CartesianPosition.Y = currentPosition.Coordinates.Y;
734     pointToSend.Position.CartesianPosition.X = currentPosition.Coordinates.X;
735     pointToSend.Position.CartesianPosition.ThetaX = start_theta[0];
```

```
736     pointToSend.Position.CartesianPosition.ThetaY = start_theta[1];
737     pointToSend.Position.CartesianPosition.ThetaZ = start_theta[2];
738     MySendBasicTrajectory(pointToSend);
739     Sleep(3000);
740
741     MyGetCartesianCommand(currentPosition);
742     pointToSend.Position.CartesianPosition.X = start[0];
743     pointToSend.Position.CartesianPosition.Y = start[1];
744     pointToSend.Position.CartesianPosition.Z = currentPosition.Coordinates.Z;
745     pointToSend.Position.CartesianPosition.ThetaX = start_theta[0];
746     pointToSend.Position.CartesianPosition.ThetaY = start_theta[1];
747     pointToSend.Position.CartesianPosition.ThetaZ = start_theta[2];
748     pointToSend.Position.Fingers.Finger1 = 5700;
749     pointToSend.Position.Fingers.Finger2 = 5700;
750     pointToSend.Position.Fingers.Finger3 = 5700;
751
752     MySendBasicTrajectory(pointToSend);
753     Sleep(3000);
754
755
756     while (true)
757     {
758         //wait for next goal after reaching the goal
759         if (Bodyflag == 1)
760         {
761             MyGetCartesianCommand(currentPosition);
762             rob_pos[0] = currentPosition.Coordinates.X;
763             rob_pos[1] = currentPosition.Coordinates.Y;
764             rob_pos[2] = currentPosition.Coordinates.Z;
765             Dtogoal = sqrt(pow(*xgoal - rob_pos[0] - momentum[0], 2) + pow
                (*ygoal - rob_pos[1] - momentum[1], 2) + pow(*zgoal - rob_pos[2]
                 - momentum[2], 2));
766         }
767
768         while (Dtogoal > 0.02)
769         {
770             if (Bodyflag == 1)
771             {
772                 //reset PF
773                 GUrep_bnd[0] = 0;
774                 GUrep_bnd[1] = 0;
775                 GUrep_bnd[2] = 0;
776                 GUrep_obs[0] = 0;
777                 GUrep_obs[1] = 0;
778                 GUrep_obs[2] = 0;
779
780                 //virtual position temp : position moved by momentum
781                 MyGetCartesianCommand(currentPosition);
782                 rob_pos[0] = currentPosition.Coordinates.X;
783                 rob_pos[1] = currentPosition.Coordinates.Y;
784                 rob_pos[2] = currentPosition.Coordinates.Z;
785
786                 temp[0] = rob_pos[0] + rate*momentum[0];
787                 temp[1] = rob_pos[1] + rate*momentum[1];
788                 temp[2] = rob_pos[2] + rate*momentum[2];
789
```

```
790                 //goal update
791                 if ((abs(currentPosition.Coordinates.X - *xgoal) > 0.3)
792                     || (abs(currentPosition.Coordinates.Y - *ygoal) > 0.8)
793                     || (abs(currentPosition.Coordinates.Z - *zgoal) > 0.3)
794                     )
795                 {
796                     std::cout << "You moved too fast!" << endl;
797                 }
798                 else
799                 {
800                     goal[0] = *xgoal;
801                     goal[1] = *ygoal;
802                     goal[2] = *zgoal;
803                 }
804 
805                 goal[0] = *xgoal;
806                 goal[1] = *ygoal;
807                 goal[2] = *zgoal;
808 
809 
810                 //boundary PF at temp
811                 for (int i = 0; i < 3; i++)
812                 {
813                     D = min(abs(bnd[0][i] - temp[i]), abs(bnd[1][i] - temp
                    [i]));
814                     Cons = Nu*(1.0 / ObsTh - 1.0 / D)*pow(1.0 / D, 2); //
                    negative
815                     DtoCenter = sqrt(pow(bnd_center[0] - temp[0], 2) + pow
                    (bnd_center[1] - temp[1], 2) + pow(bnd_center[2] - temp
                    [2], 2));
816                     if (D <= ObsTh)
817                     {
818                         GUrep_bnd[i] = Cons*((bnd_center[i] - temp[i]) /
                    DtoCenter);
819                     }
820                 }
821 
822                 //obstacles PF at temp
823                 for (int i = 0; i < obsnum; i++)
824                 {
825                     D = sqrt(pow(obs[i][0] - temp[0], 2) + pow(obs[i][1] -
                    temp[1], 2) + pow(obs[i][2] - temp[2], 2)) - obs[i][3];
826                     if (D <= ObsTh)
827                     {
828                         DtoCenter = sqrt(pow(obs[i][0] - temp[0], 2) + pow(obs
                    [i][1] - temp[1], 2) + pow(obs[i][2] - temp[2], 2));
829                         Cons = Nu*(1.0 / ObsTh - 1.0 / D)*pow(1.0 / D, 2); //
                    negative
830                         GUrep_obs[0] = GUrep_obs[0] + Cons*((temp[0] - obs[i]
                    [0]) / DtoCenter); // x direction
831                         GUrep_obs[1] = GUrep_obs[1] + Cons*((temp[1] - obs[i]
                    [1]) / DtoCenter); // y direction
832                         GUrep_obs[2] = GUrep_obs[2] + Cons*((temp[2] - obs[i]
                    [2]) / DtoCenter); // z direction
833                     }
834                 }
```

```
835
836                 GUrep[0] = GUrep_bnd[0] + GUrep_obs[0];
837                 GUrep[1] = GUrep_bnd[1] + GUrep_obs[1];
838                 GUrep[2] = GUrep_bnd[2] + GUrep_obs[2];
839
840                 GUatt[0] = Kappa * (temp[0] - goal[0]);
841                 GUatt[1] = Kappa * (temp[1] - goal[1]);
842                 GUatt[2] = Kappa * (temp[2] - goal[2]);
843
844                 gradient[0] = -GUrep[0] - GUatt[0];
845                 gradient[1] = -GUrep[1] - GUatt[1];
846                 gradient[2] = -GUrep[2] - GUatt[2];
847
848                 norm_gradient = sqrt(pow(gradient[0], 2) + pow(gradient[1], 2)
                      + pow(gradient[2], 2));
849
850                 //momentum(delta pos) = rate*previous momentum + PF at temp
851                 momentum[0] = rate * momentum[0] + stepsize*gradient[0] /
                      norm_gradient;
852                 momentum[1] = rate * momentum[1] + stepsize*gradient[1] /
                      norm_gradient;
853                 momentum[2] = rate * momentum[2] + stepsize*gradient[2] /
                      norm_gradient;
854                 norm_momentum = sqrt(pow(momentum[0], 2) + pow(momentum[1], 2)
                      + pow(momentum[2], 2));
855                 momentum[0] = stepsize * momentum[0] / norm_momentum;
856                 momentum[1] = stepsize * momentum[1] / norm_momentum;
857                 momentum[2] = stepsize * momentum[2] / norm_momentum;
858
859                 //send the robot to next pos
860                 pointToSend.Position.CartesianPosition.X = rob_pos[0] +
                      momentum[0];
861                 pointToSend.Position.CartesianPosition.Y = rob_pos[1] +
                      momentum[1];
862                 pointToSend.Position.CartesianPosition.Z = rob_pos[2] +
                      momentum[2];
863                 pointToSend.Position.CartesianPosition.ThetaX =
                      currentPosition.Coordinates.ThetaX;
864                 pointToSend.Position.CartesianPosition.ThetaY =
                      currentPosition.Coordinates.ThetaY;
865                 pointToSend.Position.CartesianPosition.ThetaZ =
                      currentPosition.Coordinates.ThetaZ;
866
867
868                 Dtogoal = sqrt(pow(goal[0] - rob_pos[0] - momentum[0], 2) +
                      pow(goal[1] - rob_pos[1] - momentum[1], 2) + pow(goal[2] -
                      rob_pos[2] - momentum[2], 2));
869                 numloop = numloop + 1;
870                 MySendBasicTrajectory(pointToSend);
871
872                 std::cout << numloop << endl;
873                 std::cout << "rob X : " << rob_pos[0] << "  rob Y : " <<
                      rob_pos[1] << "     rob Z : " << rob_pos[2] << endl;
874                 std::cout << "delta X : " << momentum[0] << "   delta Y : " <<
                       momentum[1] << "   delta Z : " << momentum[2] << endl;
875                 std::cout << "goal X : " << goal[0] << "    goal Y : " << goal
```

```
                    [1] << "    goal Z : " << goal[2] << endl << endl;
876
877                 Sleep(80);
878             }
879         }
880     }
881
882     flag = false;
883     hand_t.join();
884
885     result = (*MyCloseAPI)();
886     astra::terminate();
887     FreeLibrary(commandLayer_handle);
888
889     return programResult;
890 }
```