1. Introduction

Perceptual computing is the next step in human-computer interaction. It encompasses technologies that sense and understand the physical environment including gestures, voice recognition, facial recognition, motion tracking, and environment reconstruction. Advanced Intel® RealSense™ cameras F200 and R200 are at the forefront of the perceptual computing frontier. Depth sensing capabilities allow the F200 and R200 to reconstruct the 3D environment and track a device’s motion relative to the environment. The combination of environment reconstruction and motion tracking enables augmented reality experiences where virtual assets are seamlessly intertwined with reality.

While the Intel RealSense cameras can provide the data to power augmented reality applications, it is up to developers to create immersive experiences. One method of bringing an environment to life is through the use of autonomous agents. Autonomous agents are entities that act independently using artificial intelligence. The artificial intelligence defines the operational parameters and rules by which the agent must abide. The agent responds dynamically in real time to its environment, so even a simple design can result in complex behavior.

Autonomous agents can exist in many forms; though, for this discussion, the focus will be restricted to agents that move and navigate. Examples of such agents include non-player characters (NPCs) in video games and birds flocking in an educational animation. The goals of the agents will vary depending on the application, but the principles of their movement and navigation are common across all.

The intent of this article is to provide an introduction to autonomous navigation and demonstrate how it’s used in augmented reality applications. An example is developed that uses the Intel RealSense camera R200 and the Unity* 3D Game Engine. It is best to have some familiarity with the Intel® RealSense™ SDK and Unity. For information on integrating the Intel RealSense SDK with Unity, refer to: “Game Development with Unity* and Intel® RealSense™ 3D Camera” and “First look: Augmented Reality in Unity with Intel® RealSense™ R200.”

2. Autonomous Navigation

Agent-based navigation can be handled in a number of ways ranging from simple to complex, both in terms of implementation and computation. A simple approach is to define a path for the
agent to follow. A waypoint is selected, then the agent moves in a straight line towards it. While easy to implement, the approach has several problems. Perhaps the most obvious: what happens if a straight path does not exist between the agent and the waypoint (Figure 1)?

![Figure 1](image1.png)

**Figure 1.** An agent moves along a straight path towards the target, but the path can become blocked by an obstacle. Note: This discussion applies to navigation in both 2D and 3D spaces, but 2D is used for illustrative purposes.

More waypoints need to be added to route around obstacles (Figure 2).

![Figure 2](image2.png)

**Figure 2.** Additional waypoints are added to allow the agent to navigate around obstacles.

On bigger maps with more obstacles, the number of waypoints and paths will often be much larger. Furthermore, a higher density of waypoints (Figure 3) will allow for more efficient paths (less distance traveled to reach the destination).
Figure 3. As maps grow larger, the number of waypoints and possible paths increases significantly.

A large number of waypoints necessitates a method of finding a path between non-adjacent waypoints. This problem is referred to as pathfinding. Pathfinding is closely related to graph theory and has applications in many fields besides navigation. Accordingly, it is a heavily researched topic, and many algorithms exist that attempt to solve various aspects of it. One of the most prominent pathfinding algorithms is A*. In basic terms, the algorithm traverses along adjacent waypoints towards the desired destination and builds a map of all waypoints it visits and the waypoints connected to them. Once the destination is reached, the algorithm calculates a path using its generated map. An agent can then follow along the path. A* does not search the entire space, so the path is not guaranteed to be optimal. It is computationally efficient though.
A* requires full knowledge of the environment before execution. It is not able to adapt to dynamic changes such as added/removed obstacles and moving boundaries. Environments for augmented reality are dynamic by nature, since they build and change in response to the user’s movement and physical space.

For dynamic environments, it is preferable to let agents make decisions in real time, so that all current knowledge of the environment can be incorporated into the decision. Thus, a behavior framework must be defined so the agent can make decisions and act in real time. With respect to navigation, it is convenient and common to separate the behavior framework into three layers:

1. **Action Selection** is comprised of setting goals and determining how to achieve those goals. For example, a bunny will wander around looking for food, unless there is a predator nearby, in which case, the bunny will flee. State machines are useful for representing such behavior as they define the states of the agent and the conditions under which states change.

2. **Steering** is the calculation of the movement based on the current state of the agent. If the bunny is being chased by the predator, it should flee away from the predator. Steering calculates both the magnitude and direction of the movement force.

3. **Locomotion** is the mechanics through which the agent moves. A bunny, a human, a car, and a spaceship all move in different ways. Locomotion defines both how the agent moves (e.g., legs, wheels, thrusters, etc.) and the parameters of that motion (e.g., mass, maximum speed, maximum force, etc.).

Together these layers form the artificial intelligence of the agent. In Section 3, we'll show a Unity example to demonstrate the implementation of these layers. Section 4 will integrate the autonomous navigation into an augmented reality application using the R200.
3. Implementing Autonomous Navigation

This section walks through the behavior framework in a Unity scene for autonomous navigation described above from the ground up, starting with locomotion.

**Locomotion**

The locomotion of the agent is based on Newton’s laws of motions where force applied to mass results in acceleration. We will use a simplistic model with uniformly distributed mass that can have force applied in any direction to the body. To constrain the movement, the maximum force and the maximum speed must be defined (Listing 1).

```csharp
public float mass = 1f; // Mass (kg)
public float maxSpeed = 0.5f; // Maximum speed (m/s)
public float maxForce = 1f; // Maximum force (N)
```

*Listing 1. The locomotion model for the agent.*

The agent must have a rigidbody component and a collider component that are initialized on start (Listing 2). Gravity is removed from the rigidbody for simplicity of the model, but it is possible to incorporate.

```csharp
private void Start () {

    // Initialize the rigidbody
    this.rb = GetComponent<Rigidbody> ();
    this.rb.mass = this.mass;
    this.rb.useGravity = false;

    // Initialize the collider
    this.col = GetComponent<Collider> ();
}
```

*Listing 2. The rigidbody and collider components are initialized on Start().*

The agent is moved by applying force to the rigidbody in the FixedUpdate() step (Listing 3). FixedUpdate() is similar to Update(), but it is guaranteed to execute at a consistent interval (which Update() is not). The Unity engine performs the physics calculations (operations on rigidbodies) at the completion of the FixedUpdate() step.

```csharp
private void FixedUpdate () {

    Vector3 force = Vector3.forward;

    // Upper bound on force
    if (force.magnitude > this.maxForce) {
        force = force.normalized * this.maxForce;
    }
```
// Apply the force
rb.AddForce (force, ForceMode.Force);

// Upper bound on speed
if (rb.velocity.magnitude > this.maxSpeed) {
    rb.velocity = rb.velocity.normalized * this.maxSpeed;
}

Listing 3. Force is applied to rigidbody in the FixedUpdate() step. This example moves the agent forward along the Z axis.

if the magnitude of the force exceeds the maximum force of the agent, it is scaled such that its magnitude is equivalent to the maximum force (direction is preserved). The AddForce() function applies the force via numerical integration:

\[ v_1 = v_0 + \frac{f}{m} \Delta t \]


where \( v_1 \) is the new velocity, \( v_0 \) is the previous velocity, \( f \) is the force, \( m \) is the mass, and \( \Delta t \) is the time step between updates (the default fixed time step in Unity is 0.02 s). If the magnitude of the velocity exceeds the maximum speed of the agent, it is scaled such that its magnitude is equivalent to the maximum speed.

Steering

Steering calculates the force that will be supplied to the locomotion model. Three steering behaviors will be implemented: seek, arrive, and obstacle avoidance.

Seek

The Seek behavior attempts to move towards a target as fast as possible. The desired velocity of the behavior points directly at the target at maximum speed. The steering force is calculated as the difference between the desired and current velocity of the agent (Figure 5).
Figure 5. The Seek behavior applies a steering force from the current velocity to the desired velocity.

The implementation (Listing 4) first computes the desired vector by normalizing the offset between the agent and the target and multiplying it by the maximum speed. The steering force returned is the desired velocity minus the current velocity, which is the velocity of the rigidbody.

```csharp
private Vector3 Seek () {
    Vector3 desiredVelocity = (this.seekTarget.position - this.transform.position).normalized * this.maxSpeed;
    return desiredVelocity - this.rb.velocity;
}
```

Listing 4. Seek steering behavior.

The agent uses the Seek behavior by invoking Seek() when it computes the force in FixedUpdate() (Listing 5).

```csharp
private void FixedUpdate () {
    Vector3 force = Seek ();
    ...
}
```

Listing 5. Invoking Seek() in FixedUpdate().

An example of the Seek behavior in action is shown in Video 1. The agent has a blue arrow that indicates the current velocity of the rigidbody and a red arrow that indicates the steering force being applied in that time step.
Video 1. The agent initially has a velocity orthogonal to the direction of the target, so its motion follows a curve.

Arrive

The Seek behavior overshoots and oscillates around the target, because it was traveling as fast as possible to the reach target. The Arrive behavior is similar to the Seek behavior except that it attempts to come to a complete stop at the target. The “deceleration radius” parameter defines the distance from the target at which the agent will begin to decelerate. When the agent is within the deceleration radius, the desired velocity will be scaled inversely proportional to the distance between the agent and the target. Depending on the maximum force, maximum speed, and deceleration radius, it may not be able to come to a complete stop.

The Arrive behavior (Listing 6) first calculates the distance between the agent and the target. A scaled speed is calculated as the maximum speed scaled by the distance divided by the deceleration radius. The desired speed is taken as the minimum of the scaled speed and maximum speed. Thus, if the distance to the target is less than the deceleration radius, the desired speed is the scaled speed. Otherwise, the desired speed is the maximum speed. The remainder of the function performs exactly like Seek using the desired speed.

// Arrive deceleration radius (m)
public float decelerationRadius = 1f;

private Vector3 Arrive () {

    // Calculate the desired speed
    Vector3 targetOffset = this.seekTarget.position - this.transform.position;
    float distance = targetOffset.magnitude;
    float scaledSpeed = (distance / this.decelerationRadius) * this.maxSpeed;
    float desiredSpeed = Mathf.Min (scaledSpeed, this.maxSpeed);

    // Compute the steering force
    Vector3 desiredVelocity = targetOffset.normalized * desiredSpeed;
    return desiredVelocity - this.rb.velocity;
}

Listing 6. Arrive steering behavior.

Video 2. The Arrive behavior decelerates as it reaches the target.

Obstacle Avoidance
The Arrive and Seek behaviors are great for getting places, but they are not suited for handling obstacles. In dynamic environments, the agent will need to be able to avoid new obstacles that appear. The Obstacle Avoidance behavior looks ahead of the agent along the intended path and determines if there are any obstacles to avoid. If obstacles are found, the behavior calculates a force that alters the path of the agent to avoid the obstacle (Figure 6).

![Obstacle Avoidance Diagram](image.png)

**Figure 6.** When an obstacle is detected along the current trajectory, a force is returned that prevents the collision.

The implementation of Obstacle Avoidance (Listing 7) uses a sphercast to detect collisions. The sphercast casts a sphere along the current velocity vector of the rigidbody and returns a RaycastHit for every collision. The sphere originates from the center of the agent and has a radius equal to the radius of the agent's collider plus an “avoidance radius” parameter. The avoidance radius allows the user to define the clearance around the agent. The cast is limited to traveling the distance specified by the “forward detection” parameter.

```csharp
// Avoidance radius (m). The desired amount of space between the agent and obstacles.
public float avoidanceRadius = 0.03f;
// Forward detection radius (m). The distance in front of the agent that is checked for obstacles.
public float forwardDetection = 0.5f;

private Vector3 ObstacleAvoidance () {
    Vector3 steeringForce = Vector3.zero;

    // Cast a sphere, that bounds the avoidance zone of the agent, to detect obstacles
```
Listing 7. Obstacle Avoidance steering behavior.

The spherecast returns an array of RaycastHit objects. A RaycastHit contains information about a collision including the distance to the collision and the normal of the surface that was hit. The normal is a vector that is orthogonal to the surface. Accordingly, it can be used to direct the agent away from the collision point. The magnitude of the force is determined by scaling the maximum force inversely proportional to the distance from the collision. The forces for each collision are summed, and the result produced is the total steering force for a single time step.

Separate behaviors can be combined together to create more complex behaviors (Listing 8). Obstacle Avoidance is only useful when it works in tandem with other behaviors. In this example (Video 3), Obstacle Avoidance and Arrive are combined together. The implementation combines the behaviors simply by summing their forces. More complex schemes are possible that incorporate heuristics to determine priority weighting on forces.

Listing 8. Arrive and Obstacle Avoidance are combined by summing their forces.

https://www.youtube.com/watch?v=p_3-zU5GBEI
Video 3. The agent combines two behaviors, Arrive and Obstacle Avoidance.

Action Selection

Action selection is the high level goal setting and decision making of the agent. Our agent implementation already incorporates a simple action selection model by combining the Arrive and Obstacle Avoidance behaviors. The agent attempts to arrive at the target, but it will adjust its trajectory when obstacles are detected. The “Avoidance Radius” and “Forward Detection” parameters of Obstacle Avoidance define when action will be taken.

4. Integrating the R200

Now that the agent is capable of navigating on its own, it is ready to be incorporated into an augmented reality application.

The following example is built on top of the “Scene Perception” example that comes with the Intel RealSense SDK. The application will build a mesh using Scene Perception, and the user will be able to set and move the target on the mesh. The agent will then navigate around the generated mesh to reach the target.

Scene Manager

A scene manager script initializes the scene and handles the user input. Touch up (or mouse click release, if the device does not support touch) is the only input. A raycast from the point of the touch determines if the touch is on the generated mesh. The first touch spawns the target on the mesh; the second touch spawns the agent; and every subsequent touch moves the position of the target. A state machine handles the control logic (Listing 9).

```csharp
// State machine that controls the scene:
// Start => SceneInitialized => TargetInitialized => AgentInitialized
private enum SceneState { SceneInitialized, TargetInitialized, AgentInitialized };
private SceneState state = SceneState.SceneInitialized;  // Initial scene state.

private void Update () {

    // Trigger when the user "clicks" with either the mouse or a touch up gesture.
    if (Input.GetMouseButtonUp (0)) {
        TouchHandler ();
    }
}

private void TouchHandler () {
```
RaycastHit hit;

// Raycast from the point touched on the screen
if (Physics.Raycast(Camera.main.ScreenPointToRay(Input.mousePosition), out hit)) {

    // Only register if the touch was on the generated mesh
    if (hit.collider.gameObject.name == "meshPrefab(Clone)") {

        switch (this.state) {
            case SceneState.SceneInitialized:
                SpawnTarget (hit);
                this.state = SceneState.TargetInitialized;
                break;
            case SceneState.TargetInitialized:
                SpawnAgent (hit);
                this.state = SceneState.AgentInitialized;
                break;
            case SceneState.AgentInitialized:
                MoveTarget (hit);
                break;
            default:
                Debug.LogError("Invalid scene state.");
                break;
        }
    }
}

Listing 9. The touch handler and state machine for the example application.

The Scene Perception feature generates lots of small meshes. These meshes typically have less than 30 vertices. The positioning of the vertices is susceptible to variance, which results in some meshes being angled differently than the surface it resides on. If an object is placed on top of the mesh (e.g., a target or an agent), the object will be oriented incorrectly. To circumvent this issue, the average normal of the mesh is used instead (Listing 10).

private Vector3 AverageMeshNormal(Mesh mesh) {

    Vector3 sum = Vector3.zero;

    // Sum all the normals in the mesh
    for (int i = 0; i < mesh.normals.Length; i++){
        sum += mesh.normals[i];
    }

    // Return the average
    return sum / mesh.normals.Length;
}

Listing 10. Calculate the average normal of a mesh.
Building the Application

All code developed for this example is available on Github.

The following instructions integrate the scene manager and agent implementation into an Intel® RealSense™ application.

1. Open the “RF_ScenePerception” example in the Intel RealSense SDK folder “RSSDK\framework\Unity”.
2. Download and import the AutoNavAR Unity package.
3. Open the “RealSenseExampleScene” in the “Assets/AutoNavAR/Scenes/” folder.
4. Build and run on any device compatible with an Intel RealSense camera R200.

Video 4. The completed integration with the Intel® RealSense™ camera R200.

5. Going Further with Autonomous Navigation

We developed an example that demonstrates an autonomous agent in an augmented reality application using the R200. There are several ways in which this work could be extended to improve the intelligence and realism of the agent.

The agent had a simplified mechanical model with uniform mass and no directional movement restrictions. A more advanced locomotive model could be developed that distributes mass non-uniformly and constrains the forces applied to the body (e.g., a car with differing acceleration and braking forces, a spaceship with main and side thrusters). More accurate mechanical models will result in more realistic movement.

Craig Reynolds was the first to extensively discuss steering behaviors in the context of animation and games. The Seek, Arrive, and Obstacle Avoidance behaviors that were demonstrated in the example find their origins in his work. Reynolds described other behaviors including Flee, Pursuit, Wander, Explore, Obstacle Avoidance, and Path Following. Group behaviors are also discussed including Separation, Cohesion, and Alignment. “Programming Game AI by Example” by Mat Buckland is another useful resource that discusses the implementation of these behaviors as well as a number of other related concepts including state machines and pathfinding.

In the example, both the Arrive and Obstacle Avoidance steering behaviors are applied to the agent simultaneously. Any number of behaviors can be combined in this way to create more complex behaviors. For instance, a flocking behavior is built from the combination of Separation, Cohesion, and Alignment. Combining behaviors can sometimes produce unintuitive results. It is worth experimenting with types of behaviors and their parameters to discover new possibilities.
Additionally, some pathfinding techniques are intended for use in dynamic environments. The D* algorithm is similar to A*, but it can update the path based on new observations (e.g., added/removed obstacles). D* Lite operates in the same fashion as D* and is simpler to implement. Pathfinding can also be used in conjunction with steering behaviors by setting the waypoints and allowing steering to navigate to those points.

While action selection has not been discussed in this work, it is widely studied in game theory. Game theory investigates the mathematics behind strategy and decision making. It has applications in a many fields including economics, political science, and psychology to name a few. With respect to autonomous agents, game theory can inform how and when decisions are made. “Game Theory 101: The Complete Textbook” by William Spaniel is a great starting point and has a companion YouTube series.

6. Conclusion

An arsenal of tools exist that you can use to customize the movement, behavior, and actions of agents. Autonomous navigation is particularly well suited for dynamic environments, such as those generated by Intel RealSense cameras in augmented reality applications. Even simple locomotion models and steering behaviors can produce complex behavior without prior knowledge of the environment. The multitude of available models and algorithms allows for the flexibility to implement an autonomous solution for nearly any application.

Notices

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.