Movement

Loạt bài hướng dẫn về điều khiển di chuyển nhân vật.

- Trượt một Sphere

Trượt một Sphere

Đây là phần đầu tiên của loạt tutorial về cách điều khiển **movement** của một character. Cụ thể, chúng ta sẽ làm sphere trượt dựa trên player input.

Tutorial này được thực hiện bằng Unity 2019.2.9f1. Nội dung giả định rằng bạn đã có kiến thức cơ bản với Unity.

Kiểm soát vị trí (Controlling Position)

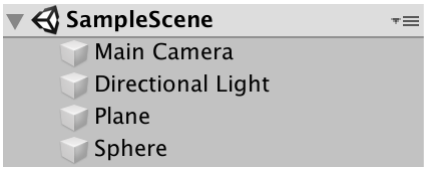
Nhiều game xoay quanh một character cần di chuyển để hoàn thành mục tiêu nào đó. Nhiệm vụ của người chơi là dẫn dắt character. Game action cho phép bạn điều khiển trực tiếp bằng cách lái character, thường là nhấn phím hoặc xoay stick. Game point-and-click yêu cầu bạn chỉ định position đích rồi character tự động di chuyển tới đó. Game programming yêu cầu bạn viết các instruction để character thực thi. Và còn nhiều kiểu khác.

Trong loạt tutorial này, chúng ta sẽ tập trung vào cách điều khiển một character trong 3D action game. Trước tiên hãy bắt đầu đơn giản bằng việc làm một sphere trượt trên một hình chữ nhật phẳng nhỏ. Khi đã nắm vững, chúng ta có thể tăng độ phức tạp trong tương lai.

Thi?t l?p Scene

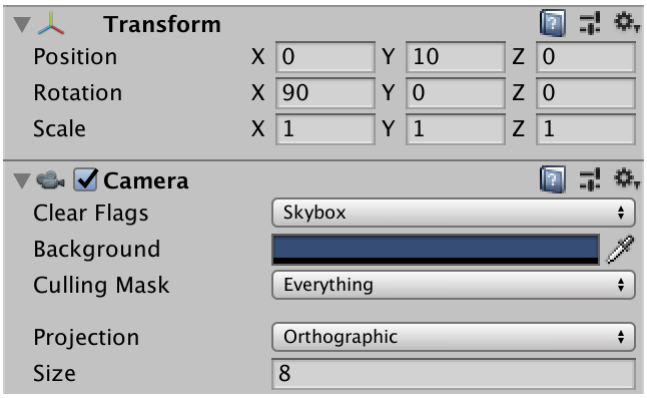
Bắt đầu với một 3D project mới theo cấu hình mặc định. Lúc này chúng ta chưa cần gì từ Package Manager, bạn có thể dùng bất kỳ render pipeline nào.

`SampleScene` mặc định có một camera và một directional light; chúng ta sẽ giữ lại cả hai. Tạo một plane đại diện cho mặt đất và thêm một sphere, đều đặt tại gốc tọa độ. Sphere mặc định có radius 0.5, vì vậy hãy đặt tọa độ Y của nó thành 0.5 để trông như đang nằm trên ground plane.



Scene hierarchy.

Chúng ta chỉ giới hạn movement 2D trên mặt đất, vì vậy hãy đặt camera phía trên plane và hướng xuống để có góc nhìn tốt về play area trong Game window. Đồng thời đặt `Projection` mode thành `Orthographic`. Cách này loại bỏ perspective, giúp quan sát movement 2D mà không bị biến dạng.

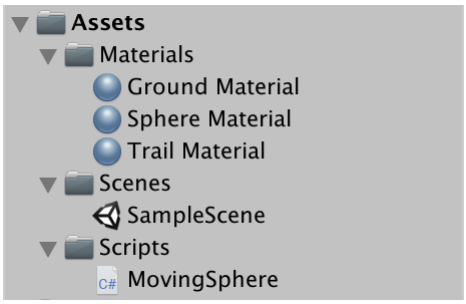


Orthographic camera nhìn từ trên xuống.

Thứ duy nhất còn làm vướng tầm nhìn là shadow của sphere. Hãy bỏ nó bằng cách đặt `Shadow Type` thành `None`, hoặc `No Shadows` tùy phiên bản Unity.

Light không tạo shadow.

Tạo material cho ground và sphere rồi cấu hình theo ý bạn. Tôi đặt sphere màu đen và ground màu xám nhạt trầm. Chúng ta cũng sẽ trực quan hóa movement bằng một trail, vì vậy hãy tạo thêm material cho nó. Tôi sẽ dùng một unlit material màu đỏ nhạt. Cuối cùng, chúng ta cần một `MovingSphere` script để triển khai movement.

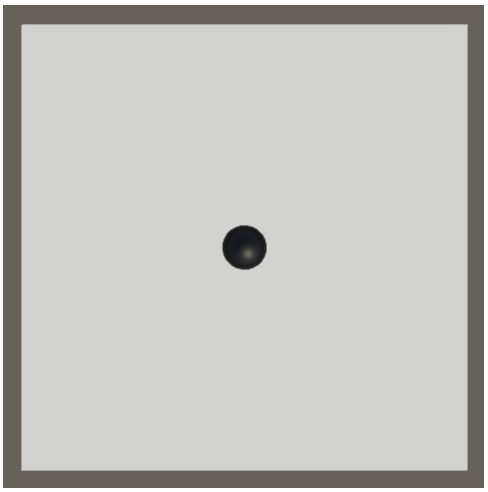


Các asset trong project.

Script có thể bắt đầu dưới dạng một extension rỗng của `MonoBehaviour`.

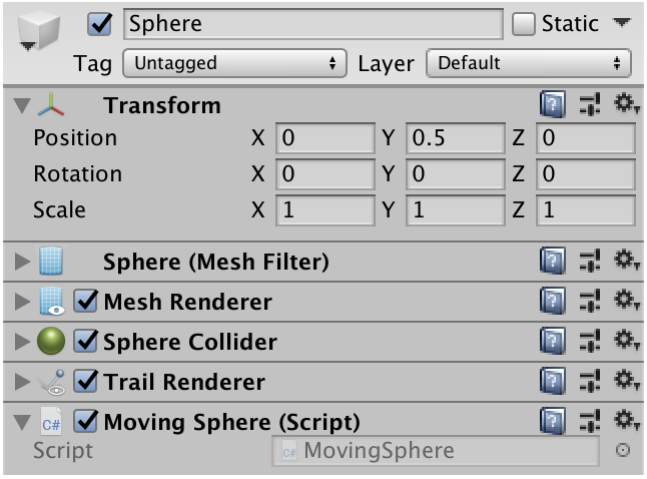
```
using UnityEngine;

public class MovingSphere : MonoBehaviour { }
```



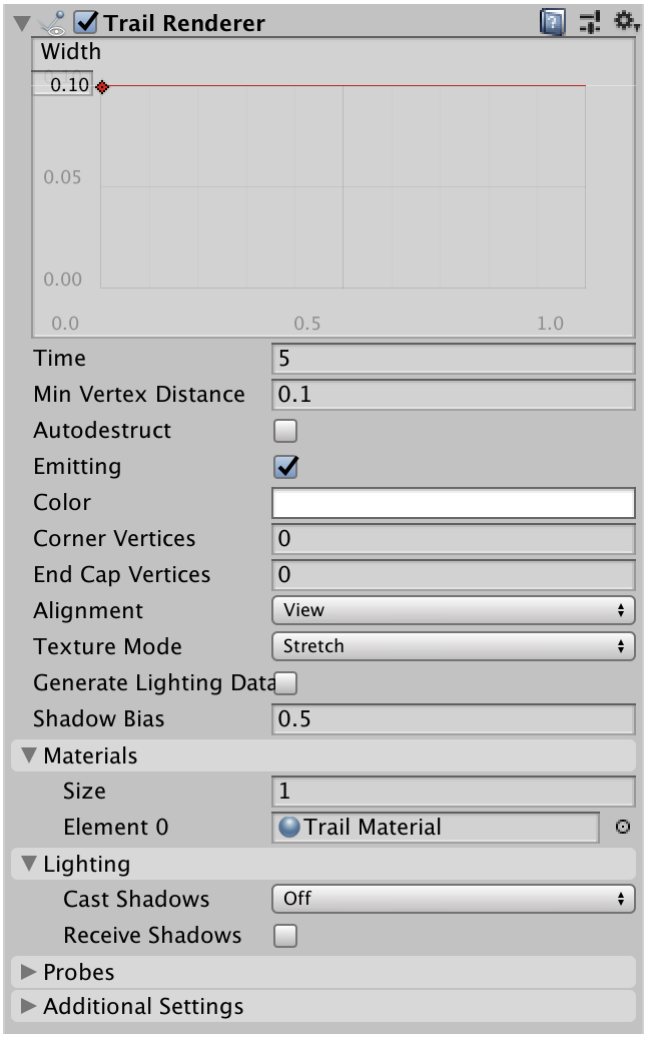
Game view.

Thêm cả `TrailRenderer` và component `MovingSphere` của chúng ta vào sphere. Giữ nguyên mọi thứ còn lại.



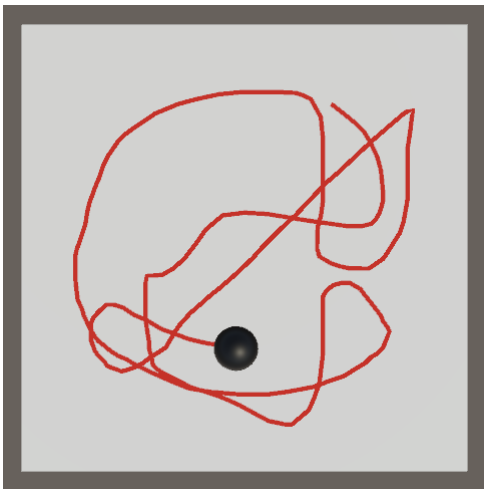
Sphere object cùng các component.

Gán trail material cho phần tử đầu tiên và duy nhất trong `Materials` array của component `TrailRenderer`. Nó không cần cast shadow, dù điều này không quan trọng vì chúng ta đã tắt shadow. Ngoài ra, hãy giảm `Width` từ 1.0 xuống một giá trị hợp lý hơn như 0.1 để tạo một đường mảnh.



Trail renderer.

Dù chưa viết code cho movement, chúng ta vẫn có thể xem trước bằng cách vào play mode rồi di chuyển sphere trong Scene window.



Movement trail.

??c Player Input

Để di chuyển sphere, chúng ta phải đọc các input command của người chơi. Việc này được thực hiện trong `Update` method của `MovingSphere`. Player input là 2D nên có thể lưu trong một `Vector2` variable. Ban đầu, ta đặt cả component X và Y của nó bằng 0, sau đó dùng chúng để đặt position của sphere trong plane XZ. Vì vậy component Y của input trở thành component Z của position. Position Y vẫn bằng 0.

```
using UnityEngine;

public class MovingSphere : MonoBehaviour {

 void Update () {
  Vector2 playerInput;
  playerInput.x = 0f;
  playerInput.y = 0f;
  transform.localPosition = new Vector3(playerInput.x, 0.5f, playerInput.y);
 }
}
```

Cách đơn giản nhất để lấy directional input từ người chơi là gọi `Input.GetAxis` với tên của một axis. Unity mặc định định nghĩa cả input axis `Horizontal` và `Vertical`; bạn có thể kiểm tra chúng trong phần `Input` của Project Settings. Ta sẽ dùng giá trị horizontal cho X và vertical cho Y.

```
playerInput.x = Input.GetAxis("Horizontal");
  playerInput.y = Input.GetAxis("Vertical");
```

Cấu hình mặc định liên kết các axis này với phím mũi tên và WASD. Các giá trị input cũng được tinh chỉnh để phím hoạt động gần giống controller stick. Bạn có thể điều chỉnh các thiết lập này theo ý muốn, nhưng tôi giữ nguyên mặc định.

Dùng phím mũi tên hoặc WASD.

Cả hai axis còn có một định nghĩa thứ hai, liên kết chúng với input từ joystick hoặc controller stick bên trái. Cách này cho input mượt hơn nhiều, nhưng tôi sẽ dùng phím cho mọi animation ngoại trừ animation tiếp theo.

Dùng controller stick.

Normalize Input Vector

Các axis trả về 0 khi ở trạng thái nghỉ và −1 hoặc 1 tại hai cực. Vì ta dùng input để đặt position của sphere nên position bị giới hạn trong một hình chữ nhật có cùng range. Ít nhất điều này đúng với key input vì các phím độc lập với nhau. Với stick, các dimension liên kết với nhau và thông thường ta bị giới hạn ở khoảng cách tối đa 1 tính từ origin theo mọi direction, do đó position nằm trong một hình tròn.

Ưu điểm của controller input là độ dài tối đa của input vector luôn bằng 1 bất kể direction. Vì vậy movement có thể nhanh như nhau theo mọi direction. Với phím thì không như vậy: giá trị tối đa là 1 khi nhấn một phím nhưng là √2 khi nhấn đồng thời hai phím, nghĩa là movement theo đường chéo

nhanh nhất.

Giá trị tối đa khi dùng phím là √2 theo định lý Pythagoras. Các giá trị axis xác định độ dài hai cạnh góc vuông, còn vector tổng hợp là cạnh huyền. Vì vậy magnitude của input vector là `?(x²+y²)`.

Ta có thể đảm bảo độ dài vector không bao giờ vượt quá 1 bằng cách chia input vector cho magnitude của nó. Kết quả luôn là unit vector, trừ khi độ dài ban đầu bằng 0, khi đó kết quả không xác định. Quá trình này được gọi là normalize một vector. Ta có thể thực hiện bằng cách gọi `Normalize` trên vector; vector sẽ tự scale và trở thành zero vector nếu kết quả lẽ ra không xác định.

```
playerInput.x = Input.GetAxis("Horizontal");
  playerInput.y = Input.GetAxis("Vertical");
  playerInput.Normalize();
```

Key input đã được normalize.

Gi?i h?n Input Vector

Luôn normalize input vector sẽ giới hạn position trên đường tròn, trừ khi input ở trạng thái neutral thì ta kết thúc tại origin. Các đường nối giữa origin và đường tròn biểu diễn một frame đơn lẻ khi hình tròn nhảy từ tâm ra đường tròn hoặc quay lại.

Kiểu input tất-cả-hoặc-không này đôi khi hữu ích, nhưng hãy cho phép cả mọi position bên trong đường tròn. Ta chỉ điều chỉnh input vector khi magnitude của nó vượt quá 1. Một cách thuận tiện là gọi static method `Vector2.ClampMagnitude` thay cho `Normalize`, truyền vào vector và giá trị tối đa 1. Kết quả là một vector được giữ nguyên hoặc được scale xuống mức tối đa đã cung cấp.

```
//playerInput.Normalize();
  playerInput = Vector2.ClampMagnitude(playerInput, 1f);
```

Key input đã được giới hạn.

Kiểm soát vận tốc (Controlling Velocity)

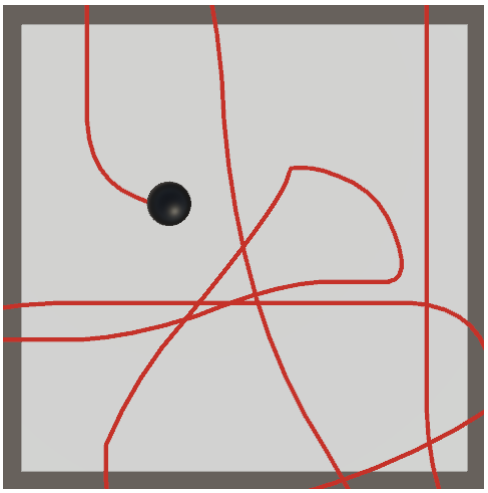
Cho đến giờ, chúng ta dùng trực tiếp input để đặt position của sphere. Điều này có nghĩa là khi input vector `i` thay đổi thì position `p` của sphere lập tức đổi thành cùng giá trị. Do đó `p=i`. Đây không

phải motion đúng nghĩa mà là teleportation. Cách tự nhiên hơn để điều khiển sphere là xác định position tiếp theo `p?` bằng cách cộng displacement vector `d` vào position cũ `p?`, do đó `p?=p?+d`.

Relative Movement

Khi dùng `d=i` thay cho `p=i`, mối quan hệ giữa input và position trở nên ít trực tiếp hơn. Điều này loại bỏ các ràng buộc lên position vì sau update đầu tiên, position trở thành tương đối với chính nó thay vì với origin. Vì thế position được mô tả bởi infinite iterative sequence `p???=p?+d` với `p?` được định nghĩa là starting position.

```
Vector3 displacement = new Vector3(playerInput.x, 0f, playerInput.y);
  transform.localPosition += displacement;
```



Relative movement.

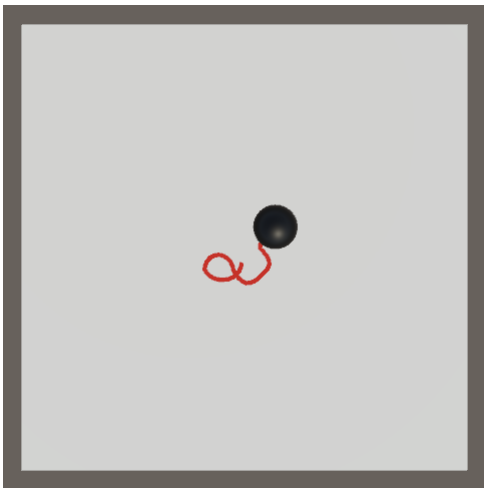
Velocity

Sphere của chúng ta thực sự có thể đi đến bất kỳ đâu, nhưng nó di chuyển nhanh đến mức khó điều khiển. Nguyên nhân là input vector được cộng vào ở mỗi update. Frame rate càng cao, sphere càng đi nhanh. Để kết quả nhất quán, ta không muốn frame rate ảnh hưởng đến input. Nếu dùng input cố định, ta muốn displacement cố định bất kể frame rate có thể dao động.

Trong trường hợp này, một frame biểu diễn một duration: lượng time `t` đã trôi qua giữa thời điểm bắt đầu frame trước và frame hiện tại; ta có thể truy cập giá trị này qua `Time.deltaTime`. Do đó displacement thực tế là `d=it` và ta đã giả định sai rằng `t` là hằng số.

Displacement được đo bằng Unity unit, giả định tương ứng với một meter. Nhưng ta nhân input với một duration tính bằng second. Để thu được meter, input phải được đo bằng meter per second. Vì vậy input vector biểu diễn velocity: `v=i` và `d=vt`.

```
Vector3 velocity = new Vector3(playerInput.x, 0f, playerInput.y);
  Vector3 displacement = velocity * Time.deltaTime;
  transform.localPosition += displacement;
```



Kiểm soát velocity độc lập với frame rate.

Speed

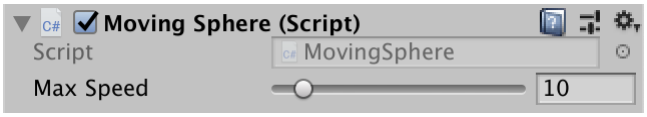
Input vector tối đa có magnitude bằng 1, tương ứng với velocity một meter per second, tức 3.6 kilometer per hour hay khoảng 2.24 mile per hour. Mức này không nhanh lắm.

Ta có thể tăng maximum velocity bằng cách scale input vector. Scale factor biểu diễn maximum speed, tức velocity không có direction. Thêm một `maxSpeed` field có attribute `SerializeField` với giá trị mặc định 10 và gắn cho nó attribute `Range` với range chẳng hạn 1–100.

```
[SerializeField, Range(0f, 100f)]
 float maxSpeed = 10f;
```

Nhân input vector với max speed để tìm desired velocity.

```
Vector3 velocity =
   new Vector3(playerInput.x, 0f, playerInput.y) * maxSpeed;
```



Max speed được đặt thành 10.

Acceleration

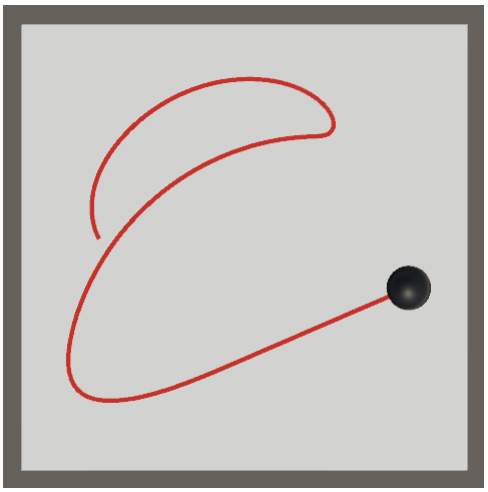
Vì chúng ta kiểm soát trực tiếp velocity nên nó có thể thay đổi tức thời. Chỉ có filtering do input system áp dụng làm thay đổi chậm lại đôi chút. Trong thực tế, velocity không thể thay đổi ngay lập tức. Một thay đổi cần effort và time, tương tự như thay đổi position. Rate of change của velocity được gọi là acceleration `a`, dẫn đến `v???=v?+at` với `v?` là zero vector. Deceleration đơn giản là acceleration ngược hướng với velocity hiện tại nên không cần xử lý đặc biệt.

Hãy xem điều gì xảy ra nếu dùng input vector để kiểm soát acceleration thay vì trực tiếp kiểm soát velocity. Việc này yêu cầu theo dõi velocity hiện tại, vì vậy hãy lưu nó trong một field.

```
Vector3 velocity;
```

Input vector giờ xác định acceleration trong `Update`, nhưng tạm thời vẫn hãy nhân nó với `maxSpeed` và diễn giải giá trị này thành max acceleration. Sau đó cộng nó vào velocity trước khi tính displacement.

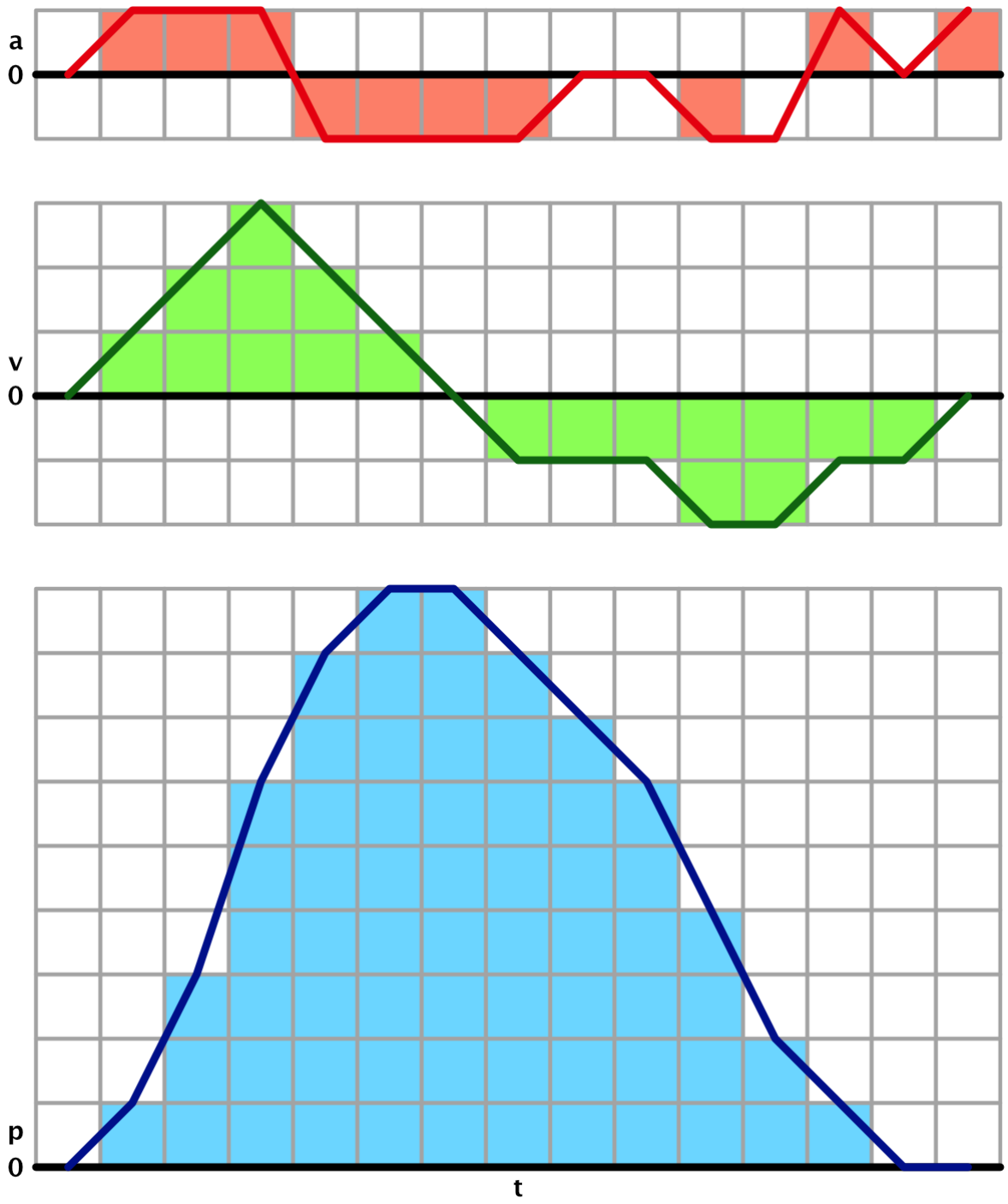
```
Vector3 acceleration =
   new Vector3(playerInput.x, 0f, playerInput.y) * maxSpeed;
  velocity += acceleration * Time.deltaTime;
  Vector3 displacement = velocity * Time.deltaTime;
```



Velocity thay đổi mượt.

Desired Velocity

Kiểm soát acceleration thay vì velocity tạo ra motion mượt hơn nhiều, nhưng cũng làm giảm khả năng kiểm soát sphere. Nó giống như ta đang lái xe thay vì đi bộ. Hầu hết game cần kiểm soát velocity trực tiếp hơn, vì vậy hãy quay lại cách tiếp cận đó. Tuy nhiên, việc áp dụng acceleration thực sự tạo ra motion mượt hơn.



Acceleration thay đổi velocity, rồi velocity thay đổi position.

Ta có thể kết hợp hai cách tiếp cận bằng việc trực tiếp kiểm soát target velocity và áp dụng acceleration lên actual velocity cho đến khi nó khớp desired velocity. Sau đó có thể tinh chỉnh responsiveness của sphere bằng cách điều chỉnh maximum acceleration. Hãy thêm một serializable field cho giá trị đó.

```
[SerializeField, Range(0f, 100f)]
 float maxAcceleration = 10f;
```

Trong `Update`, giờ ta dùng input vector để xác định desired velocity và không còn điều chỉnh velocity theo cách cũ.

```
Vector3 desiredVelocity =
   new Vector3(playerInput.x, 0f, playerInput.y) * maxSpeed;
  //velocity += acceleration * Time.deltaTime;
```

Thay vào đó, trước tiên ta tìm maximum speed change bằng cách nhân max acceleration với `t`. Đây là mức velocity tối đa có thể thay đổi trong update này.

```
Vector3 desiredVelocity =
   new Vector3(playerInput.x, 0f, playerInput.y) * maxSpeed;
  float maxSpeedChange = maxAcceleration * Time.deltaTime;
```

Trước tiên chỉ xét component X của velocity. Nếu nó nhỏ hơn desired value thì cộng maximum change.

```
float maxSpeedChange = maxAcceleration * Time.deltaTime;
  if (velocity.x < desiredVelocity.x) {
   velocity.x += maxSpeedChange;
  }
```

Điều này có thể gây overshoot; ta ngăn lại bằng cách lấy minimum giữa giá trị đã tăng và desired value. Có một `Mathf.Min` method có thể dùng ở đây.

```
if (velocity.x < desiredVelocity.x) {
   //velocity.x += maxSpeedChange;
   velocity.x =
    Mathf.Min(velocity.x + maxSpeedChange, desiredVelocity.x);
  }
```

Ngược lại, velocity có thể lớn hơn desired value. Khi đó ta trừ max change rồi lấy maximum giữa kết quả và desired value thông qua `Mathf.Max`.

```
if (velocity.x < desiredVelocity.x) {
   velocity.x =
    Mathf.Min(velocity.x + maxSpeedChange, desiredVelocity.x);
  }
  else if (velocity.x > desiredVelocity.x) {
   velocity.x =
    Mathf.Max(velocity.x - maxSpeedChange, desiredVelocity.x);
  }
```

Ta cũng có thể thực hiện toàn bộ việc này bằng `Mathf.MoveTowards` method tiện dụng, truyền vào current value, desired value và maximum allowed change. Thực hiện riêng cho cả component X và Z.

```
float maxSpeedChange = maxAcceleration * Time.deltaTime;
  //if (velocity.x < desiredVelocity.x) {
  // velocity.x =
  //  Mathf.Min(velocity.x + maxSpeedChange, desiredVelocity.x);
  //}
  //else if (velocity.x > desiredVelocity.x) {
  // velocity.x =
  //  Mathf.Max(velocity.x - maxSpeedChange, desiredVelocity.x);
  //}
  velocity.x =
   Mathf.MoveTowards(velocity.x, desiredVelocity.x, maxSpeedChange);
```

```
  velocity.z =
   Mathf.MoveTowards(velocity.z, desiredVelocity.z, maxSpeedChange);
```

Cả max speed và acceleration đều được đặt thành 10.

Giờ ta có thể tinh chỉnh maximum acceleration để đạt được điểm cân bằng mong muốn giữa motion mượt và responsiveness.

Giới hạn vị trí (Constraining Position)

Bên cạnh việc kiểm soát velocity của character, một phần quan trọng trong game là giới hạn nơi nó có thể đến. Scene đơn giản của chúng ta có một plane đại diện cho mặt đất. Hãy buộc sphere luôn nằm trên plane.

Gi? bên trong hình vuông

Thay vì dùng chính plane, ta chỉ cần tạo allowed area thành một serializable field của sphere. Có thể dùng một `Rect` struct value cho mục đích này. Gán default value khớp với plane mặc định bằng cách gọi constructor method với −5 cho hai argument đầu và 10 cho hai argument cuối. Chúng xác định bottom-left corner và size.

```
[SerializeField]
 Rect allowedArea = new Rect(-5f, -5f, 10f, 10f);
```

Ta giới hạn sphere bằng cách giới hạn new position trước khi gán nó cho `transform.localPosition`. Vì vậy trước tiên hãy lưu nó trong một variable.

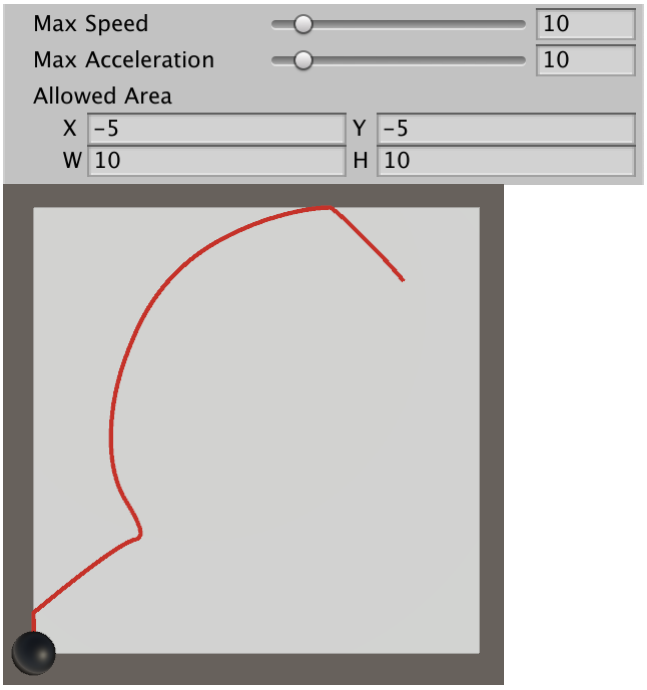
```
//transform.localPosition += displacement;
  Vector3 newPosition = transform.localPosition + displacement;
  transform.localPosition = newPosition;
```

Ta có thể gọi `Contains` trên allowed area để kiểm tra một point có nằm bên trong hoặc trên edge hay không. Nếu new position không thỏa điều kiện, hãy đặt nó về current position, qua đó triệt tiêu movement trong update này.

```
Vector3 newPosition = transform.localPosition + displacement;
  if (!allowedArea.Contains(newPosition)) {
   newPosition = transform.localPosition;
  }
  transform.localPosition = newPosition;
```

Khi truyền một `Vector3` vào `Contains`, method sẽ kiểm tra tọa độ XY, không đúng trong trường hợp này. Vì vậy hãy truyền một `Vector2` mới chứa tọa độ XZ.

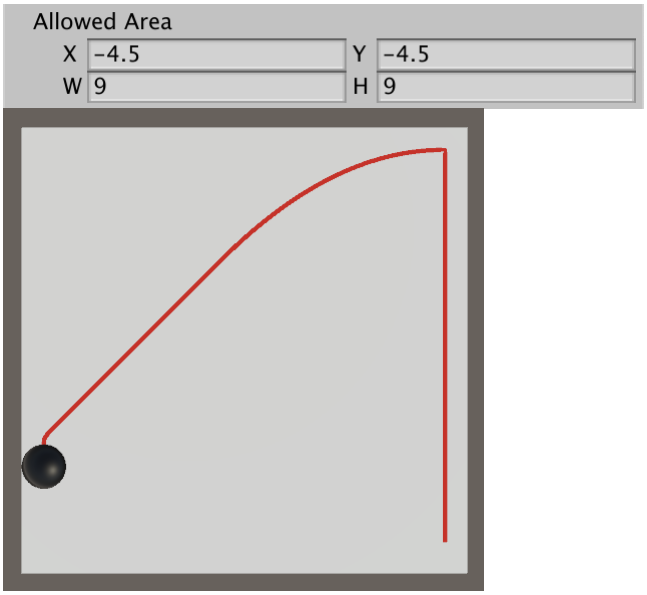
```
if (!allowedArea.Contains(new Vector2(newPosition.x, newPosition.z))) {
   newPosition = transform.localPosition;
  }
```



Dừng lại tại các edge của plane.

Sphere giờ không thể thoát ra ngoài và sẽ dừng khi cố làm vậy. Kết quả bị giật vì movement bị bỏ qua trong một số frame, nhưng ta sẽ xử lý ngay sau đây. Trước hết, hãy lưu ý sphere có thể đi đến khi nằm ngay trên edge của plane. Nguyên nhân là ta chỉ giới hạn position mà không tính đến radius. Sẽ đẹp hơn nếu toàn bộ sphere luôn nằm trong allowed area. Ta có thể sửa code để tính đến radius, nhưng cách khác là chỉ cần thu nhỏ allowed area. Như vậy là đủ cho scene đơn giản này.

Dịch corner của area lên 0.5 và giảm size đi 1 ở cả hai dimension.



Dừng lại khi chạm edge của plane.

??t Position chính xác

Ta có thể loại bỏ movement bị giật bằng cách clamp new position vào allowed area thay vì bỏ qua nó. Có thể thực hiện bằng cách gọi `Mathf.Clamp` với một value cùng minimum và maximum được phép. Dùng property `xMin` và `xMax` của area cho X, còn `yMin` và `yMax` cho Z.

```
if (!allowedArea.Contains(new Vector2(newPosition.x, newPosition.z))) {
   //newPosition = transform.localPosition;
   newPosition.x =
    Mathf.Clamp(newPosition.x, allowedArea.xMin, allowedArea.xMax);
   newPosition.z =
    Mathf.Clamp(newPosition.z, allowedArea.yMin, allowedArea.yMax);
  }
```

Bám vào các edge.

Lo?i b? Velocity

Sphere giờ có vẻ bám vào các edge. Sau khi chạm edge, ta trượt dọc theo nó nhưng phải mất một lúc mới có thể rời khỏi edge. Nguyên nhân là velocity của sphere vẫn hướng vào edge. Ta phải đổi direction bằng acceleration hướng ra khỏi edge; việc này mất nhiều hay ít time tùy vào max acceleration.

Nếu sphere là một ball và edge của area là một wall thì khi chạm wall, nó phải dừng lại. Điều này thực sự xảy ra. Nhưng nếu wall đột ngột biến mất, ball sẽ không lấy lại velocity trước đó. Momentum đã mất, energy được truyền đi trong collision và có thể gây damage. Vì vậy phải loại bỏ velocity khi chạm edge. Tuy nhiên sphere vẫn có thể trượt dọc edge, nên chỉ component velocity hướng về edge đó cần bị loại bỏ.

Để đặt component velocity phù hợp về 0, ta phải kiểm tra xem có vượt bounds theo cả hai hướng của mỗi dimension hay không. Lúc này ta cũng có thể tự clamp position vì đang thực hiện đúng các kiểm tra giống như `Mathf.Clamp` và `Contains`.

```
//if (!allowedArea.Contains(new Vector2(newPosition.x, newPosition.z))) {
   //newPosition.x =
   // Mathf.Clamp(newPosition.x, allowedArea.xMin, allowedArea.xMax);
   //newPosition.z =
   // Mathf.Clamp(newPosition.z, allowedArea.yMin, allowedArea.yMax);
  //}
  if (newPosition.x < allowedArea.xMin) {
   newPosition.x = allowedArea.xMin;
   velocity.x = 0f;
  }
  else if (newPosition.x > allowedArea.xMax) {
   newPosition.x = allowedArea.xMax;
   velocity.x = 0f;
  }
```

```
  if (newPosition.z < allowedArea.yMin) {
   newPosition.z = allowedArea.yMin;
   velocity.z = 0f;
  }
  else if (newPosition.z > allowedArea.yMax) {
   newPosition.z = allowedArea.yMax;
   velocity.z = 0f;
  }
```

Không còn bám vào edge.

N?y

Velocity không phải lúc nào cũng bị loại bỏ trong collision. Nếu sphere là một ball nảy hoàn hảo thì nó sẽ đảo direction trong dimension tương ứng. Hãy thử làm như vậy.

```
if (newPosition.x < allowedArea.xMin) {
   newPosition.x = allowedArea.xMin;
   velocity.x = -velocity.x;
  }
  else if (newPosition.x > allowedArea.xMax) {
   newPosition.x = allowedArea.xMax;
   velocity.x = -velocity.x;
  }
  if (newPosition.z < allowedArea.yMin) {
   newPosition.z = allowedArea.yMin;
   velocity.z = -velocity.z;
  }
  else if (newPosition.z > allowedArea.yMax) {
   newPosition.z = allowedArea.yMax;
   velocity.z = -velocity.z;
  }
```

Nảy khỏi các edge.

Sphere giờ giữ lại momentum và chỉ đổi direction khi chạm wall. Nó chậm lại đôi chút vì sau khi nảy, velocity không còn khớp với desired velocity. Để có cú nảy tốt nhất, người chơi phải điều chỉnh input ngay lập tức.

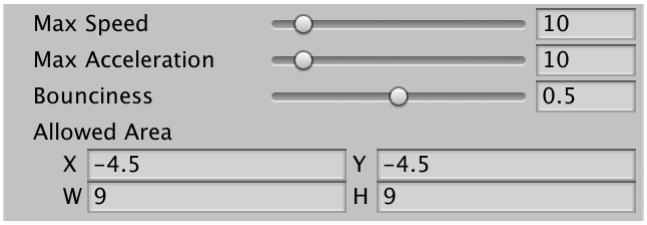
Bounciness

Không cần giữ lại toàn bộ velocity khi đảo chiều. Một số vật nảy nhiều hơn vật khác. Vì vậy hãy cho phép cấu hình bằng cách thêm một `bounciness` field, mặc định là 0.5 với range 0–1. Nhờ đó sphere có thể nảy hoàn hảo, không nảy chút nào hoặc ở bất kỳ mức nào giữa hai trạng thái.

```
[SerializeField, Range(0f, 1f)]
 float bounciness = 0.5f;
```

Đưa bounciness vào new velocity value khi chạm edge.

```
if (newPosition.x < allowedArea.xMin) {
   newPosition.x = allowedArea.xMin;
   velocity.x = -velocity.x * bounciness;
  }
  else if (newPosition.x > allowedArea.xMax) {
   newPosition.x = allowedArea.xMax;
   velocity.x = -velocity.x * bounciness;
  }
  if (newPosition.z < allowedArea.yMin) {
   newPosition.z = allowedArea.yMin;
   velocity.z = -velocity.z * bounciness;
  }
  else if (newPosition.z > allowedArea.yMax) {
   newPosition.z = allowedArea.yMax;
   velocity.z = -velocity.z * bounciness;
  }
```



Bounciness được đặt thành 0.5.

Cách này không mô phỏng physics thực tế, vốn phức tạp hơn nhiều. Nhưng kết quả đã bắt đầu trông giống thật và như vậy là đủ cho hầu hết game. Ngoài ra, movement của chúng ta chưa thật chính xác. Các phép tính chỉ đúng nếu sphere chạm edge đúng vào cuối movement trong một frame. Khả năng cao điều đó không xảy ra, nghĩa là sphere lẽ ra phải lập tức di chuyển ra xa edge một chút. Trước tiên cần tính remaining time rồi dùng nó cùng new velocity trong dimension tương ứng. Tuy nhiên, điều này có thể dẫn đến lần nảy thứ hai, khiến mọi thứ phức tạp hơn. May mắn là ta không cần độ chính xác như vậy để tạo ảo giác thuyết phục về một sphere đang nảy xung quanh.

Tutorial tiếp theo là Physics.