Kinematic Character Controller (KCC)

Introduction

A Kinematic Character Controller, KCC for short, is used to move a character within the world according to its own set of rules. Using a KCC rather than physics/force based movement allows for tighter control and snappy movement. Although those concepts are core to every game, they vary tremendously in their definition as they are related to the overall gameplay. Therefore the KCCs included in the Quantum SDK are to be considered a starting point; however, game developers will likely have to create their own in order to get the best possible results for their specific context.

Quantum comes with two pre-build KCCs, one for 2D (side-scrolling) and one for 3D movement. The API allows characters to move through terrains, climb steps, slide down slopes, and use moving platforms.

The KCCs take physics data of both static and dynamic objects into consideration when calculating the movement vectors. Objects will block and define the character's movement. Collision callbacks with the environment objects will be trigged as well.

Requirements

To use or add a KCC to an entity, the entity has to already have a *Transform* component. A *PhysicsBodycan* be used but is **not** necessary; it is generally advised against using a *PhysicsBody* with a KCC as the physics system may affect it and result in unintended movement.

If you are not familiar with Quantum's Physics yet, please review the *Physics* documentation first.

Raycasts & ShapeOverlap

raycasting, it has to also carry a *PhysicsCollider*.

Note

This page covers both the 2D and 3D KCCs.

The Character Controller Component

You can add the CharacterController component to your entity by either:

- adding the "Character Controller" component to the Entity Prototype in Unity; or,
- adding the "Character Controller" component via code.

The Character Controller 2D and 3D components attached to an Entity Prototype in the Unity Editor.

To add the Character Controller via code, follow the examples below.

C#

```
// 2D KCC
var kccConfig = FindAsset<CharacterController2DConfig>(KCC_CONFIG
var kcc = new CharacterController2D();
kcc.Init(f, kccConfig)
f.Add(entity, kcc);

// 3D KCC
var kccConfig = FindAsset<CharacterController3DConfig>(KCC_CONFIG
var kcc = new CharacterController3D();
kcc.Init(f, kccConfig)
f.Add(entity, kcc);
```

The component has to be initiliazed after being created. The available initializing options are:

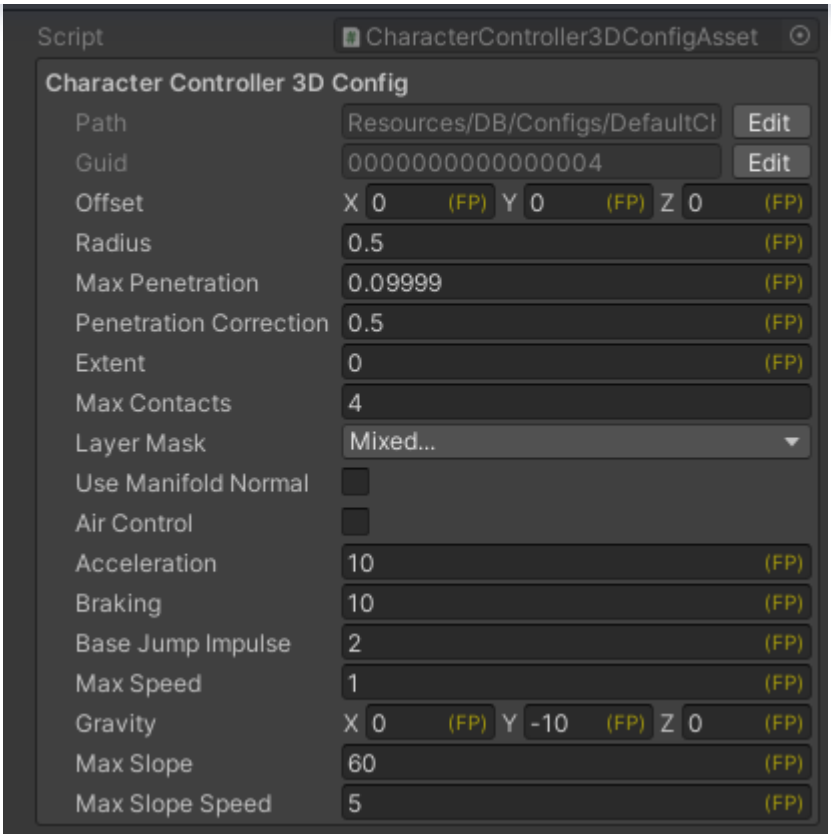
- (code) the `Init()` method *without* parameter, it will load the `DefaultCharacterController` from `Assets/Resources/DB/Configs`.
- (code) the `Init()` method *with* parameter, it will load the passed in `CharacterControllerConfig`.
- (editor) add the `CharacterControllerConfig` to the `Config` slot in the `Character Controller` component.

The Character Controller Config

Create your own KCC config asset via the context menu under `Create > Quantum > Assets > Physics > CharacterController2D/3D`.

Default Config Assets

The default 2D and 3D KCC Config assets are located inside the `Assets/Resources/DB/Configs` folder. Here is how the 3D KCC config looks like:



The DefaultCharacterController3D Config Asset.

A brief explanation into the Config fields

- **Offset** is used to define the KCC local position based into the entity position. It is commonly used to position the center of the KCC at the feet of the character. Remember: the KCC is used to *move* the character, so it does not necessarily have to encapsulate the character's whole body.
- **Radius** defines the boundaries of the character and should encompass the character horizontal size. This is used to know whether a character can move in a certain direction, a wall is blocking the movement, a step is to be climbed, or a slope to be slid on.
- **Max Penetration** smoothens the movement when a character penetrates other physics objects. If the character passes the Max Penetration, a hard fix will be applied and snap it into the correct position. Reducing this value to zero will apply all corrections fully and instantly; this may result in jagged movement.
- **Extent** defines a radius in which collisions are detected preemptively.
- **Max Contacts** is used to select the amount of contact points computed by the KCC. 1 will usually work fine and is the most performant option. If you experience jerky movement, try setting this to 2; the additional overhead is negligible.
- **Layer Mask** defines which collider layers should be taken into consideration by the physics query performed by the KCC.
- **Air Control** toggle to `True` and the KCC is able to perform movement adjustments when it not touching the ground.
- **Acceleration** defines how fast the character accelerates.

- **Max Speed** caps the character's maximal horizontal speed.
- **Gravity** applies a gravity force to the KCC.
- **Max Slope** defines the maximal angle, in degrees, the character can walk up and down.
- **Max Slope Speed** limits the speed at which the character slides down a slope when the movement type is Slope Fall instead of Horizontal Fall.

Character Controller API

The API shown below focuses on the 3D KCC. The 2D and 3D APIs are very similar though.

Properties and Fields

Each CharacterController component has these fields.

C#

```
public FP MaxSpeed { get; set;}
public FPVector3 Velocity { get; set;}
public bool Grounded { get; set;}
public FP CurrentSpeed { get;}
public AssetGUID ConfigId { get;}
```

Tip

The *MaxSpeed* is a cached value after initialization. It can therefore be modified at runtime, e.g. when performing dashes.

API

Each KCC components has the following methods:

C#

```
public void Init(FrameBase frame, CharacterController3DConfig con

// Jump
public void Jump(FrameBase frame, bool ignoreGrounded = false, FP

// Move
public void Move(FrameBase frame, EntityRef entity, FPVector3 dir

// Raw Information
public static CharacterController3DMovement ComputeRawMovement(Fr
```

The `Jump` and `Move` methods are convenient for prototyping, while `ComputeRawMovement` provides the key information for creating your own custom movement. In the example KCC's provided by Quantum, the information from `ComputeRawMovement` is used by the internal steering method `ComputeRawSteer` to compute the steering used in `Move`.

IMPORTANT: The implementations of `Jump()`, `Move()` and `ComputeRawSteer()` are presented below for fostering understanding and help create custom implementations specific to the game's requirements.

CharacterController3DMovement

`ComputeRawMovement()` computes the environmental data necessary for the steering by performing a ShapeOverlap and processing the data. The method returns a `CharacterController3DMovement` struct which can then be applied to the character movement. The movement data provided can also be used to create a custom steering implementation.

The `CharacterController3DMovement` struct holds the following information:

C#

```
public enum CharacterMovementType
{
    None, // grounded with no desired direction passed
    FreeFall, // no contacts within the Radius
    SlopeFall, // there is at least 1 ground contact within the R
    Horizontal, // there is NO ground contact, but there is at le
}
```

```
{
  public CharacterMovementType Type;

  // the surface normal of the closest unique contact
  public FPVector3 NearestNormal;

  // the average normal from all contacts
  public FPVector3 AvgNormal;

  // the normal of the closest contact that qualifies as ground
  public FPVector3 GroundNormal;

  // the surface tangent (from GroundNormal and the derived direc
  public FPVector3 Tangent;

  // surface tangent computed from closest the contact normal vs
  public FPVector3 SlopeTangent;

  // accumulated projected correction from all contacts within th
  public FPVector3 Correction;

  // max penetration of the closest contact within the Radius
  public FP Penetration;

  // uses the EXTENDED radius to assign this Boolean AND the Grou
  public Boolean Grounded;

  // number of contacts within Radius
  public int Contacts;
}
```

`ComputeRawMovement()` is used by the `Move()` method.

Jump()

This is only a reference implementation.

C#

```
public void Jump(FrameBase frame, bool ignoreGrounded = false, FP

  if (Grounded || ignoreGrounded) {

    if (impulse.HasValue)
      Velocity.Y.RawValue = impulse.Value.RawValue;
    else {
      var config = frame.FindAsset(Config);
      Velocity.Y.RawValue = config.BaseJumpImpulse.RawValue;
    }

    Jumped = true;
  }
}
```

Move()

This is only a reference implementation.

`Move()` takes the following things by taking into consideration when calculating the character's new position:

- the current position
- the direction
- the gravity
- jumps
- slopes
- and more

All these aspects can be defined in the config asset passed to the `Init()` method. This is convenient for prototyping FPS/TPS/Action games which have terrains, mesh colliders and primitives.

`ComputeRawMovement()` and create your own custom steering + movement.

C#

```
public void Move(Frame frame, EntityRef entity, FPVector3 directi
  Assert.Check(frame.Has<Transform3D>(entity));

  var transform = frame.GetPointer<Transform3D>(entity);
  var dt        = deltaTime ?? frame.DeltaTime;

  CharacterController3DMovement movementPack;
  fixed (CharacterController3D* thisKcc = &this) {
    movementPack = ComputeRawMovement(frame, entity, transform, t
  }

  ComputeRawSteer(frame, ref movementPack, dt);

  var movement = Velocity * dt;
  if (movementPack.Penetration > FP.EN3) {
    var config = frame.FindAsset<CharacterController3DConfig>(Con
    if (movementPack.Penetration > config.MaxPenetration) {
      movement += movementPack.Correction;
    } else {
      movement += movementPack.Correction * config.PenetrationCor
    }
  }

  transform->Position += movement;
}
```

ComputeRawSteer()

Steering involves computing the movement based on the position, radius and velocity of the character, and corrects the movement if necessary.

This is only a reference implementation.

the `movementPack` values from `ComputeRawMovement` and passes them to `ComputeRawSteer`.

C#

```
private void ComputeRawSteer(FrameThreadSafe f, ref CharacterCont

  Grounded = movementPack.Grounded;
  var config = f.FindAsset(Config);
  var minYSpeed = -FP._100;
  var maxYSpeed = FP._100;

  switch (movementPack.Type) {

    // FreeFall
    case CharacterMovementType.FreeFall:

      Velocity.Y -= config._gravityStrength * dt;

      if (!config.AirControl || movementPack.Tangent == default(F
        Velocity.X = FPMath.Lerp(Velocity.X, FP._0, dt * config.B
        Velocity.Z = FPMath.Lerp(Velocity.Z, FP._0, dt * config.B
      } else {
        Velocity += movementPack.Tangent * config.Acceleration *
      }

      break;

    // Grounded movement
    case CharacterMovementType.Horizontal:

      // apply tangent velocity
      Velocity += movementPack.Tangent * config.Acceleration * dt
      var tangentSpeed = FPVector3.Dot(Velocity, movementPack.Tan

      // lerp current velocity to tangent
      var tangentVel = tangentSpeed * movementPack.Tangent;
      var lerp = config.Braking * dt;
      Velocity.X = FPMath.Lerp(Velocity.X, tangentVel.X, lerp);
      Velocity.Z = FPMath.Lerp(Velocity.Z, tangentVel.Z, lerp);
```

```
    // otherwise it will jump with a lower impulse
    if (Jumped == false) {
      Velocity.Y = FPMath.Lerp(Velocity.Y, tangentVel.Y, lerp);
    }

    // clamp tangent velocity with max speed
    var tangentSpeedAbs = FPMath.Abs(tangentSpeed);
    if (tangentSpeedAbs > MaxSpeed) {
      Velocity -= FPMath.Sign(tangentSpeed) * movementPack.Tang
    }

    break;

  // Sliding due to excessively steep slope
  case CharacterMovementType.SlopeFall:

    Velocity += movementPack.SlopeTangent * config.Acceleration
    minYSpeed = -config.MaxSlopeSpeed;

    break;

  // No movement, only deceleration
  case CharacterMovementType.None:

    var lerpFactor = dt * config.Braking;

    if (Velocity.X.RawValue != 0) {
      Velocity.X = FPMath.Lerp(Velocity.X, default, lerpFactor)
      if (FPMath.Abs(Velocity.X) < FP.EN1) {
        Velocity.X.RawValue = 0;
      }
    }

    if (Velocity.Z.RawValue != 0) {
      Velocity.Z = FPMath.Lerp(Velocity.Z, default, lerpFactor)
      if (FPMath.Abs(Velocity.Z) < FP.EN1) {
        Velocity.Z.RawValue = 0;
      }
    }

    // we only lerp the vertical velocity back to 0 if the char
```

```
        Velocity.Y = FPMath.Lerp(Velocity.Y, default, lerpFactor)
        if (FPMath.Abs(Velocity.Y) < FP.EN1) {
          Velocity.Y.RawValue = 0;
        }
      }

      minYSpeed = 0;

      break;
    }

    // horizontal is clamped elsewhere
    if (movementPack.Type != CharacterMovementType.Horizontal) {
      var h = Velocity.XZ;

      if (h.SqrMagnitude > MaxSpeed * MaxSpeed) {
        h = h.Normalized * MaxSpeed;
      }

      Velocity.X = h.X;
      Velocity.Y = FPMath.Clamp(Velocity.Y, minYSpeed, maxYSpeed);
      Velocity.Z = h.Y;
    }

    // reset jump state
    Jumped = false;
  }
```

Collision Callbacks

Whenever the KCC detects intersections with colliders a callback is triggered.

C#

```
public interface IKCCCallbacks2D
{
    bool OnCharacterCollision2D(FrameBase f, EntityRef character,
    void OnCharacterTrigger2D(FrameBase f, EntityRef character, P
```

```
public interface IKCCCallbacks3D
{
    bool OnCharacterCollision3D(FrameBase f, EntityRef character,
    void OnCharacterTrigger3D(FrameBase f, EntityRef character, P
}
```

To receive the callbacks and use its information implement the corresponding `IKCCCallbacks` interface in a system.

Important Note that the collision callbacks return a Boolean value. This is allows you decide whether a collision should be ignored. Returning `false` makes the character pass through physics object it collided with.

Besides implementing the callbacks the movement methods should also pass the `IKCCCallbacks` object; below is a code snippet using the collision callbacks.

C#

```
namespace Quantum
{
  using Quantum.Core;
  using Quantum.Physics3D;

  public unsafe class SampleSystem : SystemMainThreadFilter<Sampl
  {
    public struct Filter
    {
      public EntityRef EntityRef;
      public CharacterController3D* KCC;
    }

    public bool OnCharacterCollision3D(FrameBase f, EntityRef cha
    {
      // read the collision information to decide if this should
      return true;
    }

    public void OnCharacterTrigger3D(FrameBase f, EntityRef chara
```

```
    public override void Update(Frame f, ref Filter filter)
    {
      // [...]
      // adding the IKCCCallbacks3D as the last parameter (this s
      //CharacterController3D.Move(, input->Direction, this);
      filter.KCC->Move(f, filter.EntityRef, input->Direction, thi
      // [...]
    }
  }
}
```

Back to top

photon

Circle Stack Overflow

Cloud Status

Connect

Public Discord

YouTube

Facebook

Twitter

Blog

Contact Us

Languages

English

日本語

한국어

简体中文

繁体中文