PPGA Game Project Report

Kobe Dereyne - 2DAE GD10

▼ Collision and Distances

To check for collision between 2 objects, we need to know how far they are apart.
To do so I use several methods to check distances between 2 GA Elements.

Distance between 2 Points

Checking for the distance between 2 points in PPGA is as simple as taking the norm of their join: $d(P,Q) = |P \vee Q|$

Distance between 2 Parallel Planes

Checking for the distance between 2 parallel planes in PPGA is the same as taking the vanishing norm of their meet: $d(m,n) = |m \wedge n|_\infty$

Distance between a Parallel Plane and Line

Checking for the distance between a plane and a line that are parallel to each other, we can take the vanishing norm of grade 3 of their gep product: $d(m,L) = | < m\ L > 3|_\infty$

Distance between a Point and a Plane

Checking the distance between a point and a plane in PPGA is the same as taking their join, which produces a scalar: $d(P,n) = P \vee n$

Distance between a Point and a Line

Checking the distance between a point and a plane in PPGA is the same as taking the norm of their join: $d(P,L) = |P \vee L|$

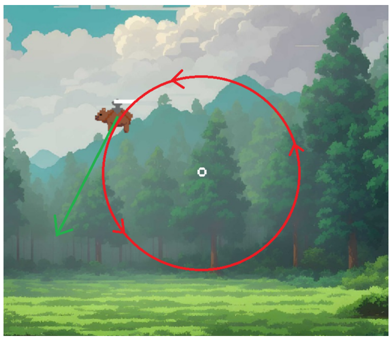
Collision

These distances between GA Elements can be used to check for collision. For example, if the distance between the position of the player (Three Blade) and an outer wall (One Blade) is smaller than a certain threshold (size of the player), we know a collision occurred.

Similarly we can check collision between 2 Three Blades (e.g. Player Position and Portal) with a certain threshold. Just as before, if their distance is smaller than said threshold, a collision occurred and we can behave accordingly.

▼ Correct Rotation according to Movement

When changing from a translation to a rotation, we want to intuitively start rotating in the correct direction.
For example, the player is moving along the green line, when suddenly we want to change to a rotation on the red line. In this situation, we would intuitively want to start rotating counter clockwise, and not clockwise.



To achieve this correct rotation we need to do some calculations.
First of all, we need the position of the player. Because the player position does not reside in
$e3$ (because we use the third dimension to store energy), we must first project it on $e3$, so we can use the 2D position of the player.

$$^{(3)}P_{3D} = P_{3D} = \mathit{Player\ Position\ in\ 3D}$$
$$P_{2D} = P = (P \cdot e3)\ e3 = \mathit{Player\ Position\ in\ 2D}$$

We also need the Three Blade around which we want to start rotating. In the game, a rotation is around a Two Blade, but in my case I store its position in a Three Blade.

$$^{(3)}Q = Q = \mathit{Rotation\ Point}$$

Next we must connect these 2 points by spanning a line between them. This can be done using the join operator, which creates a Two Blade that goes through both points. We also normalize this newly formed Two Blade.

$$^{(2)}L = L = P \vee Q$$
$$L_n = \frac{L}{|L|}$$

After this we dualize L to make use of the property that D$(L_n) \perp L_n$, this operation leaves us with $L_D = D(L_n)$

Before continuing, there is one crucial step we need to do, and that is taking the absolute value the L_D's $e12$ component. The reason for this is quite simple. Since we are working in a 2D space, both P and Q (should) have an $e03$ component that is equal to 0 (ignoring the energy stored here for the player). As a result, L_n will lie in the "conventional" XY plane, whereas L_D is perpendicular to L_n, but from the top-down view, moving away from us. Taking the absolute value of L_D's $e12$ component, we force L_D to move towards us, which will be crucial for further calculations. Further reasoning why we only do this for the $e12$ component is because the Euclidean part of a Two Blade holds its direction, with $e12$ being the "conventional z-axis".

Now only 2 steps remain. Calculating the Travel Direction of the player and the Tangent to the circle on which the player must rotate.
Let's start with the latter.
We have
L_D, which due to our previous calculations now has a direction towards us from the top-down view. We also know the following of a Line-Line Type GEP

$$LK = L \cdot K + L \times K + L \wedge K$$

But we are only interested in $< LK >_2 = L \times K$, which is known as the commutator cross and produces the common normal line of L and K. We compute this for L_D and $D(L_D)$.

$$T = < L_D \ \ D(L_D) >_2$$

Due to the orientation of L_D and $D(L_D)$, T will always have a direction in counter clockwise orientation relative to the rotation point Q when looking top-down. We also want to normalize T. And now the most important steps, calculating the Travel Direction of the player and deciding how to rotate around Q.
The Travel Line of the player is easy to calculate. We already have it's current position
P, we can simply also keep track of the player's previous position P_{prev}. When needing to calculate the travel line, we can simple join and normalize this 2 points.

$$P_{travelline} = \frac{P_{prev} \vee P}{|P_{prev} \vee P|}$$

Now we have everything to determine the correct rotation. Taking the dot of 2 lines is the following

$$L \cdot K = -(\hat{l} \cdot \hat{k}) = -||\hat{l}|| * ||\hat{l}|| * cos(\theta)$$

If we take the dot between $P_{travelline}$ and T, knowing both are normalized we get the following.

$$P_{travelline} \cdot T = -cos(\theta)$$

Therefore

$$cos(\theta) = -P_{travelline} \cdot T$$

Using $cos(\theta)$ we can determine whether or not the player's Travel Line is moving in the same or opposite direction of T. If $cos(\theta) > 0$ they have the same direction, if $cos(\theta) < 0$, they have an opposite direction. The sign of $-P_{travelline} \cdot T$ thus determines whether we need to spin clockwise or counter clockwise.

$$-P_{travelline} \cdot T < 0 => CW$$

$$-P_{travelline} \cdot T > 0 => CWW$$

▼ Clamping Points using Planes

Given a Three Blade P and a One Blade n, using the distance and projection formulas, we can actually clamp the values of P on/to n. We must define n as a plane in space such that it's orientation/normal for clamping Three Blades makes sense, meaning an n_{min} and n_{max} must face each other. For example, e_3 and $-e_3 + 3(-e_0)$ are valid planes for n, as they face each other. Because of the property we assigned, we can take the signed distance from point P to n. If this signed distance is $<= 0$, the Three Blade P is out of range and must be clamped.

Clamping P to a plane basically means projecting it onto the plane, which is done using the following formula

$$P_{||} = (P \cdot n)\, n$$

We have now "clamped" P according to a plane n, meaning if P goes out of bounds, we project it on n. In the game project, this is used to, first update the energy level of the player by translating it, and then clamp it to a plane, such that the energy level cannot to below or above a certain value.