The GameObject class contains a transform matrix, a list of CollisionRadii and flags for processing the GameObject with respect to collisions.

To move the GameObject, the appropriate function is provided a homogenous 3D vector representing a translation, the uniform factor or individual factors representing scaling, or a floating-point value representing the angle of rotation on the z-axis. The intuitive values passed-in are then converted into their matrix equivalent. The GameObject’s internal transform matrix accumulates these “delta” matrices; by the “delta” being added (in case of scaling and translation) or pre-multiplied (in the case of rotation) on a frame-by-frame basis, resulting in GameObject movement.

Note: the order in which transformations are applied in the GameObject (i.e. in the case of physics movement) is rotation, then scaling, then translation. This prevents the rotation or scale transforms from *also* applying to the translation vector and causing unexpected movement, illustrated in Fig. 1.

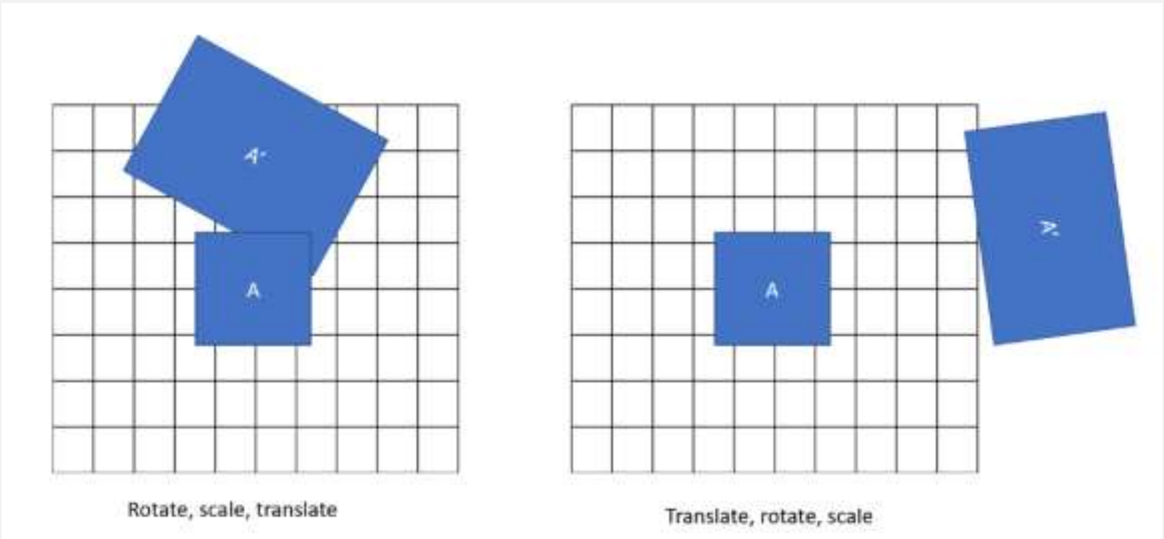

Fig. 1 - transforms for movement

The CollisionRadii struct is the crucial data structure underpinning the collision detection. A CollisionRadii stores a centre point (as centreX and centreY fields) and two synchronised lists which form a map of angles, each with a corresponding radius.

In circle-circle collisions, the boundary of an object is represented as a scalar radius (and perhaps an offset centre position). This enables fast and simple collision detection with the limitation that the boundary is a perfect circle. CollisionRadii is an addition to enable “selective-radius circle-circle collisions”: given the angle between the GameObject and the collision candidate (as measured from the GameObject’s local x-axis) we can select the radius corresponding to the clockwise-next angle in the CollisionRadii (in Fig. 2, the arc R2’ represents the boundary when the angle-to-collider is between R2 and R3).

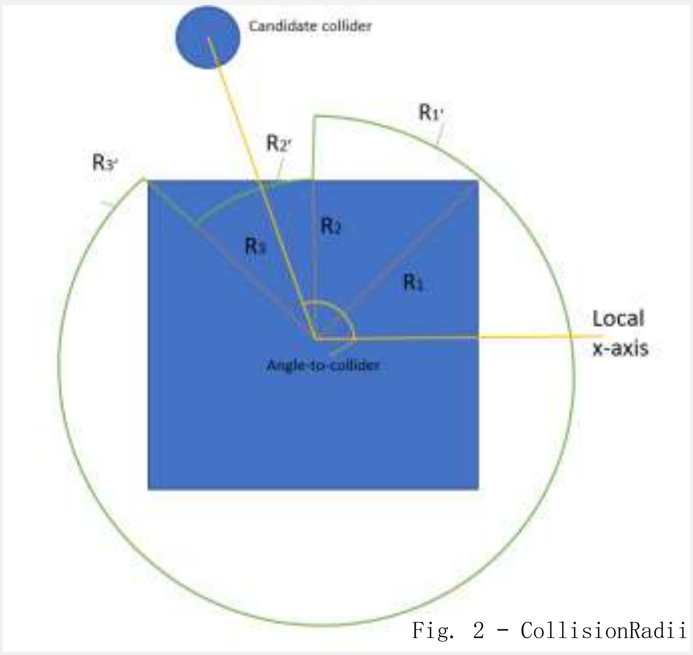

Fig. 2 - CollisionRadii

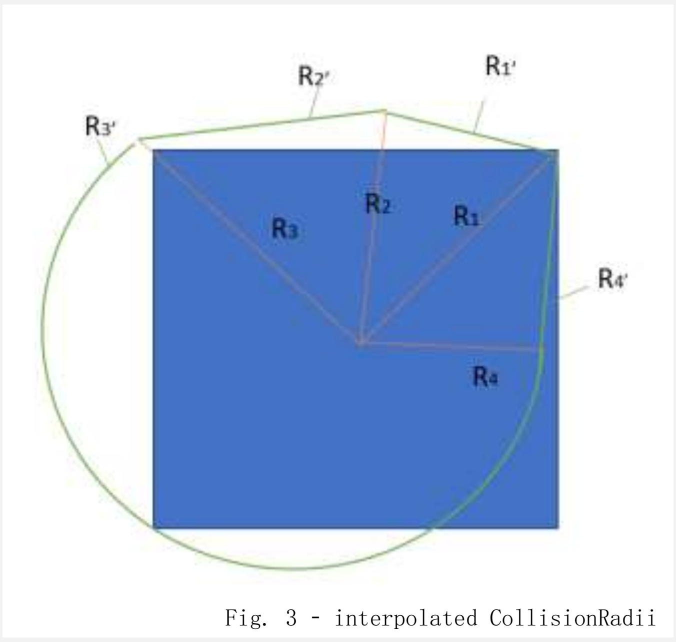

Fig. 3 - interpolated CollisionRadii

A further refinement of the CollisionRadii implementation is interpolation by linear combination of the radii either side of the angle-to-collider – this produces the effect of a linear boundary, rather than an arc, between each radius of the CollisionRadii – as in Fig. 3.

Note: CollisionRadii centre-points and the lengths of their radii are represented in a separate space, bound-space. The bounds transform matrix (internal to the GameObject) transforms the bounds independently of a GameObject’ s transform matrix. Consequently, radii and centre-points must be transformed from bounds-space to world-space for collision detection calculations.

Each GameObject has flags to control the movement and collision behaviour. Flags include physics, collisions, ghost, container and ignoreContainers. Physics enables movement calculation through dampened forces and z-axis torque. Collisions enables the GameObject in collision detection. Ghost disables effects on movement that a collision with the GameObject may have, although collisions are still detected (included for debugging and extensibility). Setting the container flag means that all collision candidates must stay within the GameObject bounds, unless a collision candidate’ s ghost or ignoreContainers flag is set, in which case the container will have no effect on the candidate.As in circle-circle collision detection; if the distance (squared) between objects is less than the *combined* radii (also squared) they collide. For containers, a collision is instead detected if the distance squared is greater than or equal to the *container’ s* radius squared.

The implementation relies on pre-computed angle-radius pairs, in this case Blender 3D was used to create the geometry and a python script created to export the values. Real-time computation or baking of these values is certainly feasible.