```
/*!*****************************************************************************
\author Jared Joyal
\par    email: jared.joyal\@digipen.edu
\copyright All content (c) 2018 DigiPen (USA) Corporation, all rights reserved.
*******************************************************************************/

/*!*****************************************************************************
\brief
  This collision system function detects collisions between two any-sided convex
  polygons using my implementation of the Separating Axis Theorem (SAT)
\param polygon1
  A structure of data regarding the first polygon
\param polygon2
  A structure of data regarding the second polygon
\param returnManifold
  A manifold to return detailed collision data. Optional in case a simple true/
  false is desired.
\return
  True if a collision was detected
*******************************************************************************/
bool PolygonVsPolygon(const Polygon &polygon1, const Polygon &polygon2,
                      std::optional<Manifold *> returnManifold)
{
  // the denominator of the calculation to measure time step
  // time step, as part of the manifold, is a float [0, 1] determining at what
  //   percentage of delta time the collision occurred
  float timeStepDenominator = 1.0f;

  // do the following code against both polygons
  for (int firstPolygon = 0; firstPolygon <= 1; ++firstPolygon)
  {
    const Polygon &thisPolygon = firstPolygon ? polygon1 : polygon2;
    const Polygon &otherPolygon = firstPolygon ? polygon2 : polygon1;

    // iterate through the normals of this polygon, running the SAT algorithm
    for (const vec2 &normal : thisPolygon.normals)
    {
      // if the other polygon is on the other side of this polygon (opposite
      //   this normal's edge), skip checking
      if (dot((otherPolygon.position - thisPolygon.position), normal) < 0)
        continue;
```

```
      // calculate time step denominator
      // time step is calculated by the amount of penetration over the total
      //   distance travelled; we can only calculate the denominator for now
      if (returnManifold)
      timeStepDenominator = length(dot(polygon1.position, normal) -
                                   dot(polygon1.lastPosition, normal)) +
                            length(dot(polygon2.position, normal) -
                                   dot(polygon2.lastPosition, normal));

      // call a helper function to check the SAT algorithm against this normal
      if (CheckThisNormal(returnManifold, normal, polygon1,
                          polygon2, !firstPolygon, timeStepDenominator))
        return false;
    }
  }

    // return if collision found
  return true;
}

/*!*****************************************************************************
\brief
  This helper function checks one normal using the SAT algorithm and stores the
  results in the optional return manifold
\return
  If no collision is found, return true. This lets us short-circuit the SAT
  algorithm. (If you can draw a straight line between two polygons, they do not
  collide.)
*******************************************************************************/
bool CheckThisNormal(std::optional<Manifold *> returnManifold, vec2 thisNormal,
                     const Polygon &polygon1, const Polygon &polygon2,
                     bool makeNormalNegative, float timeStepDenominator)
{
  // if the second polygon is being checked, the normal needs to be negated for
  //   consistent collision resolution calculations
  if (makeNormalNegative)
      thisNormal *= -1;
```

```
  // get the penetration from the SAT algorithm; if a smaller penetration is
  //   found (or it is the default value -1), use that resolution data instead
  float penetration = SAT(thisNormal, polygon1, polygon2);
  if (returnManifold && (penetration < (*returnManifold)->penetration ||
                         (*returnManifold)->penetration == -1))
  {
    (*returnManifold)->penetration = penetration;
    (*returnManifold)->timeStep = 1.0f - penetration / timeStepDenominator;
    (*returnManifold)->normal = thisNormal;
  }

  // if polygons are not colliding, short-circuit SAT algorithm
  return (penetration == 0);
}

/*!*****************************************************************************
\brief
  This helper function implements the hard math stuff of the SAT algorithm
\return
  Amount of penetration between two polygons on custom axis (parallel to normal)
*******************************************************************************/
float SAT(vec2 normal, const Polygon &polygon1, const Polygon &polygon2)
{
  // project all points to 1D along this normal; find the minimum and maximum
  //   1D values of both polygons
  float polygon1Min, polygon1Max;
  float polygon2Min, polygon2Max;

  // do the following code against both polygons
  for (int firstPolygon = 0; firstPolygon <= 1; ++firstPolygon)
  {
    const Polygon &thisPolygon = firstPolygon ? polygon1 : polygon2;
    float &thisPolygonMin = firstPolygon ? polygon1Min : polygon2Min;
    float &thisPolygonMax = firstPolygon ? polygon1Max : polygon2Max;
```

```
    // flag to determine if this is the first point checked in the upcoming loop
    bool firstPoint = true;
    // check all points of this polygon
    for (const vec2 &point : thisPolygon.points)
    {
      // project this point to 1D along the normal
      float thisPoint1D = dot(point, normal);
      // if this is the first point checked, set min and max to its 1D value
      if (firstPoint)
      {
        thisPolygonMin = thisPolygonMax = thisPoint1D;
        firstPoint = false;
      }
      else
      {
        // if this is not the first point, save the min and max values thus far
        thisPolygonMin = min(thisPolygonMin, thisPoint1D);
        thisPolygonMax = max(thisPolygonMax, thisPoint1D);
      }
    }
  }

  // check if there is a space between the max of one polygon and the min of the
  //   other; if so, we can draw a line between the polygons perpendicular to
  //   this axis and there is no collision
  if (polygon1Min < polygon2Min || polygon2Max < polygon1Min)
    return 0.0f;

  // calculate the penetration of the two polygons, taking the minimum value of
  //   the two possible cases (depending on in which order the polygons are)
  return min(polygon1Max - polygon2Min, polygon2Max - polygon1Min);
}
```