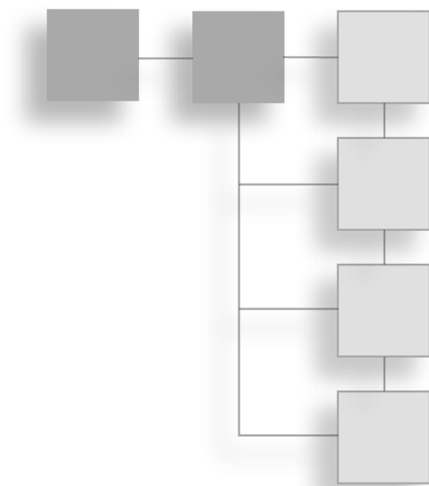
CHAPTER 18



Board Games

Board games are another of those categories of games that have mass appeal, and many of the most popular board games have been around so long that copyright issues are no longer a concern, so they can be a great source of inspiration for your original titles.

In this chapter, you will learn how to:

- Implement simple computer opponents to enhance single-player game play
- Simultaneously manage the state of multiple on-screen game objects
- Add simple, yet effective animations and visual effects to your game
- Use threading to keep animations going while processing background tasks
- Track and display a player's score using a bitmap font

The Design

In this chapter, we will develop a simple version of a popular board game that originated in England around 1880. Reversi gained wide-spread popularity in England towards the end of the 19th century, and it experienced a resurgence in popularity in the 1970s, when it was refined into its modern incarnation and banded with a new moniker—Othello.

In our version of the game, the action takes place on a 12-by-12 grid. Each player starts with two pieces, placed in the center of the grid. One point is awarded for each piece a player has on the grid. Players must place their piece on the board according to the following rules:

- The grid cell into which the piece is placed must be empty.
- The grid cell into which the piece is placed must be adjacent to one of the opponent's pieces.
- The player's new piece, along with an existing piece, must "surround" at least one piece of the opponent, in a straight line (horizontally, vertically, or diagonally).

Players take turns placing pieces on the board in an attempt to capture their opponent's pieces and maximize the number of grid cells that are occupied by their own pieces. When there is no legal move for a player, he forfeits his turn and play passes to his opponent. If neither player can make a legal move (as is the case when the grid is full), the game is over. The player with the most pieces on the board wins.

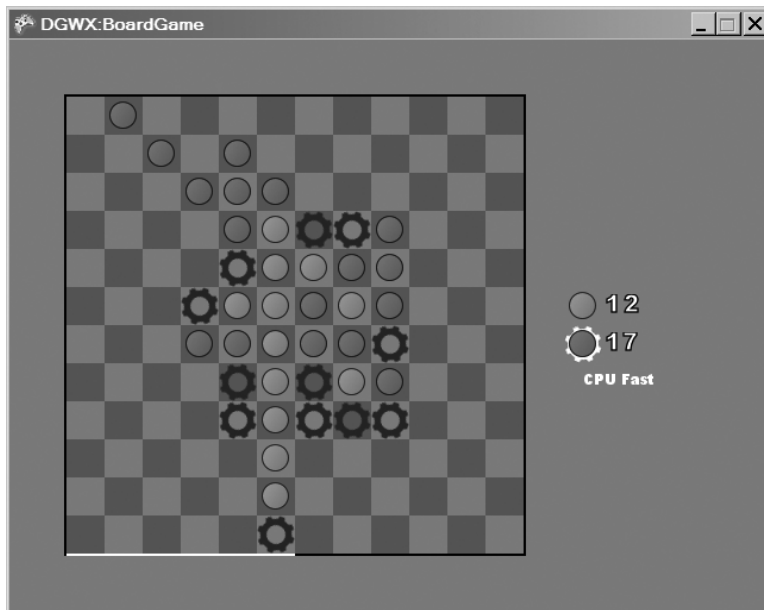


Figure 18.1
A screen shot of the game that we will develop in this chapter. The CPU opponent is considering his options, and has evaluated just over half of the possible moves.

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 407

The Architecture

The basic design for this game is fairly simple. The game grid starts out empty and is gradually filled, one piece at a time. As pieces are added to the grid, we need to make sure that the additions satisfy the game rules. The basic components to handle these tasks are detailed in the following text.

The GameBoard

The `GameBoard` class contains a two-dimensional array of `GamePiece` objects. The dimensions of this array are determined by the `GridWidth` and `GridHeight` member variables. The `GameBoard` class is also responsible for making sure that any moves that are made by a player (human or CPU) conform to the rules of the game. This is done by determining the complete list of valid moves (stored in the `ValidMoves` array) whenever a game turn begins. Only moves in this list are allowed.

Determining Valid Moves

The `UpdateListOfValidMoves` method creates a list of all the grid cell locations where the current player may legally place his new game piece. New pieces may only be placed on the game board if they meet the three requirements of the game rules: the cell is empty, the cell is adjacent to an opponent's piece, and the new piece (along with an existing piece) will surround at least one of the opponent's pieces.

```
// list of valid moves for this player and board
public List<Point> ValidMoves = new List<Point>();
public void UpdateListOfValidMoves()
{
    // off-grid, start with invalid cursor position
    m_Cursor = -Vector2.One;

    // clear current valid moves
    Vector2 position = Vector2.Zero;
    ValidMoves.Clear();

    // don't bother scanning if no one's playing
    if (CurrentPlayer == Player.None) return;

    // reset scores
    int score1 = 0;
    int score2 = 0;
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 408


```
    for (int y = 0; y < GridHeight; y++) // for each row
    {
        position.Y = y;
        for (int x = 0; x < GridWidth; x++) // for each column
        {
            position.X = x;
            if (Grid[x, y].Owner == Player.None)
            {
                // rules for valid move: 1) empty cell, 2) next to
                // opponent's piece, 3) scan straight-line reveals
                // piece owned by current player
                if (Scan(position, CursorDirection.North) ||
                    Scan(position, CursorDirection.NorthEast) ||
                    Scan(position, CursorDirection.East) ||
                    Scan(position, CursorDirection.SouthEast) ||
                    Scan(position, CursorDirection.South) ||
                    Scan(position, CursorDirection.SouthWest) ||
                    Scan(position, CursorDirection.West) ||
                    Scan(position, CursorDirection.NorthWest))
                {
                    ValidMoves.Add(new Point(x, y));
                }
            }
            else if (Grid[x, y].Owner == Player.One)
            {
                score1++;
            }
            else
            {
                score2++;
            }
        }
    }

    // if there's at least one valid move, locate the cursor on
    // the first valid move in the list
    if (ValidMoves.Count > 0)
    {
        MoveCursorTo(ValidMoves[0]);
    }

    // update the NumberSprites with the new scores
    m_ScoreOne.Value = score1;
    m_ScoreTwo.Value = score2;
}
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 409

The first two rules are easy enough to check, but the `UpdateListOfValidMoves` method relies on the (first overloaded) `Scan` method to look for cells that satisfy the third rule. This version of the `Scan` method just examines the pieces that occupy the game board, without changing any state information.

```
// scan without marking
private bool Scan(Vector2 position, Vector2 direction)
{
    return Scan(position, direction, false);
}
```

Relative Cell Locations

To make the code a little easier to follow, directions are expressed as instances of the `Vector2` structure, which contain a value that can be added to a cell location to determine its neighbor. By doing this, our code can refer to the relative location of [–1,+1] as `SouthWest`, making the logic easier to follow.

```
// simple "constants" for scanning grid cells
public sealed class CursorDirection
{
    public static readonly Vector2 North     = new Vector2(  0, -1);
    public static readonly Vector2 NorthEast = new Vector2(  1, -1);
    public static readonly Vector2 East      = new Vector2(  1,  0);
    public static readonly Vector2 SouthEast = new Vector2(  1,  1);
    public static readonly Vector2 South     = new Vector2(  0,  1);
    public static readonly Vector2 SouthWest = new Vector2( -1,  1);
    public static readonly Vector2 West      = new Vector2( -1,  0);
    public static readonly Vector2 NorthWest = new Vector2( -1, -1);
}
```

Navigating the List of Valid Moves

The two overloaded `MakeMove` methods and the two `MoveCursorTo` methods serve as the gatekeepers to the `ValidMoves` array and prevent players from selecting invalid moves. The `NextValidMove` and `PreviousValidMove` methods help the game logic use gamepad events from the player to select moves from the list of valid moves.

```
// move cursor to the next valid move
public void NextValidMove()
{
    Point current = new Point((int)m_Cursor.X, (int)m_Cursor.Y);
    int index = ValidMoves.IndexOf(current);
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 410


```
        index++;
        if (index >= ValidMoves.Count) index = 0;
        MoveCursorTo(index);
    }

    // move cursor to the previous valid move
    public void PreviousValidMove()
    {
        Point current = new Point((int)m_Cursor.X, (int)m_Cursor.Y);
        int index = ValidMoves.IndexOf(current);
        index--;
        if (index < 0) index = ValidMoves.Count - 1;
        MoveCursorTo(index);
    }

    // the current cursor location in grid units
    private Vector2 m_Cursor = Vector2.Zero;

    // move cursor to specified location
    public void MoveCursorTo(Point location)
    {
        m_Cursor = new Vector2(location.X, location.Y);
    }

    // move cursor to location indicated by index into ValidMoves array
    public void MoveCursorTo(int index)
    {
        if (index >= 0 && index < ValidMoves.Count)
        {
            m_Cursor =
                new Vector2(ValidMoves[index].X, ValidMoves[index].Y);
        }
    }
```

The MakeMove and Scan Methods

The `MakeMove` method actually commits the selected move to the current game board. This method assumes that the game rules were enforced when the list of valid moves was determined. Capturing opponent pieces in any direction is enough to satisfy the game rules, but this method will scan the game board in all eight directions to make sure that the current player claims all of the opponent's pieces to which he is entitled.

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 411


```
// commit currently selected cell as player's move
public void MakeMove()
{
    Point current = new Point((int)m_Cursor.X, (int)m_Cursor.Y);
    int index = ValidMoves.IndexOf(current);
    MakeMove(index);
}

// select a move from the ValidMove list, useful for CPU players
public void MakeMove(int index)
{
    if (index >= 0 && index < ValidMoves.Count)
    {
        Point move = ValidMoves[index];
        Vector2 position = new Vector2(move.X,move.Y);

        // mark current cell as belonging to the current player
        Grid[move.X, move.Y].Owner = CurrentPlayer;

        // scan each direction, if valid, claim it
        if (Scan(position, CursorDirection.North))
        {
            Scan(position, CursorDirection.North, true);
        }
        if (Scan(position, CursorDirection.NorthEast))
        {
            Scan(position, CursorDirection.NorthEast, true);
        }
        if (Scan(position, CursorDirection.East))
        {
            Scan(position, CursorDirection.East, true);
        }
        if (Scan(position, CursorDirection.SouthEast))
        {
            Scan(position, CursorDirection.SouthEast, true);
        }
        if (Scan(position, CursorDirection.South))
        {
            Scan(position, CursorDirection.South, true);
        }
        if (Scan(position, CursorDirection.SouthWest))
        {
            Scan(position, CursorDirection.SouthWest, true);
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 412


```
        }
        if (Scan(position, CursorDirection.West))
        {
            Scan(position, CursorDirection.West, true);
        }
        if (Scan(position, CursorDirection.NorthWest))
        {
            Scan(position, CursorDirection.NorthWest, true);
        }

        // move has been committed, switch players
        ToggleCurrentPlayer();
    }
}
```

The Scan Method

Once a move has been committed, the `MakeMove` method uses the second overloaded `Scan` method to claim the opponent's captured game pieces for the current player.

```
    // scan, possibly committing move to game state
    private bool Scan(Vector2 position, Vector2 direction, bool mark)
    {
        // assume caller verified that first cell is empty
        // move to next cell in scan ...
        Player other = OtherPlayer();
        position += direction;
        int count = 0;

        // ... and start matching opponent cells
        while (InBounds(position) && GetPiece(position).Owner == other)
        {
            if (mark)
            {
                Grid[(int)position.X,(int)position.Y].Owner = CurrentPlayer;
            }
            count++;
            position += direction;
        }

        // return result as boolean
        return
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 413


```
                count > 0 &&
                InBounds(position) &&
                GetPiece(position).Owner == CurrentPlayer;
    }

    // is the specified cell within the grid?
    private bool InBounds(Vector2 position)
    {
        return
            position.X >= 0 &&
            position.X < GridWidth &&
            position.Y >= 0 &&
            position.Y < GridHeight;
    }

    // get a game piece, given a vector (rather than [x,y])
    public GamePiece GetPiece(Vector2 position)
    {
        return Grid[(int)position.X, (int)position.Y];
    }
}
```

The Score Method

The `Score` method is used by the `Evaluate` method, which is called from the AI of the various computer opponents to determine the state of the potential game boards that are being examined. The score that this method calculates is also stored in the corresponding `NumberSprite` instance for each player, which is used by the heads-up display logic to render the current scores to the screen.

```
// get the score for the specified player
// rather than store the scores in member variables and try
// to keep them in sync with the display, pull Value from
// the NumberSprite directly
public int Score(Player player)
{
    long score = 0;
    switch (player)
    {
        case Player.One:
            score = m_ScoreOne.Value;
            break;
        case Player.Two:
```

```
            score = m_ScoreTwo.Value;
            break;
    }
    return (int)score;
}
```

The ToggleCurrentPlayer Method

The ToggleCurrentPlayer method is used to switch players after a valid move has been committed to the game board. It is also used by the AI of the various computer opponents to toggle between the two players as potential game boards are being evaluated. In the former case, the CpuMoveHasBeenStarted flag is set to false when the player is toggled, alerting the CPU opponent (if any) that his turn is ready to begin.

```
// switch between Player.One and Player.Two
public void ToggleCurrentPlayer()
{
    // toggle player, then regenerate list of valid moves
    CurrentPlayer = OtherPlayer();
    UpdateListOfValidMoves();

    // if there aren't any valid moves ...
    if (ValidMoves.Count == 0)
    {
        // ... switch back to the original player
        CurrentPlayer = OtherPlayer();
        UpdateListOfValidMoves();

        // if there still aren't any valid moves, the game is over
        if (ValidMoves.Count == 0)
        {
            State = GameState.GameOver;
        }
    }

    // if this player is a CPU player, note that it's ready
    // to kick off its AI processing thread
    CpuMoveHasBeenStarted = false;
}
```

In addition to managing the GamePiece objects and enforcing the game rules, the GameBoards class also manages the three artificial intelligence variants that are supported by this game (described in more detail later).

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 415

The DrawGrid Method

The `DrawGrid` method renders the game grid in a checkerboard pattern.

```
// draw the game board
private void DrawGrid(SpriteBatch batch)
{
    // big rect, draw grid border
    Rectangle borderRect = new Rectangle(
        (int)TopLeft.X - 2,
        (int)TopLeft.Y - 2,
        GridWidth * 32 + 4,
        GridHeight * 32 + 4);
    batch.Draw(Texture, borderRect, PixelRect, Color.Black);

    // size of a single grid cell
    Rectangle cellRect = new Rectangle(0, 0, 32, 32);

    // alternating colors for grid cells
    Color[] colors = {
        Color.CornflowerBlue,
        Color.RoyalBlue,
        };

    for (int y = 0; y < GridHeight; y++) // for each row
    {
        cellRect.Y = y * 32 + (int)TopLeft.Y;
        for (int x = 0; x < GridWidth; x++) // for each column
        {
            cellRect.X = x * 32 + (int)TopLeft.X;
            batch.Draw(
                Texture,
                cellRect,
                PixelRect,
                colors[(x + y) % 2]);
        }
    }
}
```

The DrawCursor Method

The `DrawCursor` method renders a rotating gear icon for every possible legal move for the current player in dark blue. If the current player is human, the cursor for the currently selected move is rendered as a spinning white gear.

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 416


```
// render valid move hints in blue, and current selection in white
private void DrawCursor(SpriteBatch batch)
{
    Point current = Point.Zero;
    Vector2 position = Vector2.Zero;
    Vector2 center = new Vector2(16, 16);

    for (int y = 0; y < GridHeight; y++) // for each row
    {
        current.Y = y;
        position.Y =
            TopLeft.Y + // top of the grid
            y * 32 +    // grid position of this cell
            center.Y;   // plus rotation offset

        for (int x = 0; x < GridWidth; x++) // for each column
        {
            current.X = x;
            position.X =
                TopLeft.X + // left of the grid
                x * 32 +    // grid position of this cell
                center.X;   // plus rotation offset

            // highlight this cell if it's selected by human player
            bool highlight = current.X == m_Cursor.X;
            highlight &= current.Y == m_Cursor.Y;
            highlight &= CurrentPlayerType() == PlayerType.Human;

            if (highlight)
            {
                batch.Draw(
                    Texture,            // cursor texture
                    position,           // cursor x, y
                    CursorRect,         // cursor source rect
                    Color.White,        // color
                    (float)m_EffectAge, // cursor rotation
                    center,             // center of cursor
                    1.0f,               // don't scale
                    SpriteEffects.None, // no effect
                    0.0f);              // topmost layer
            }
            else if(ValidMoves.Contains(current))
            {
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 417


```
                batch.Draw(
                    Texture,            // cursor texture
                    position,           // cursor x, y
                    CursorRect,         // cursor source rect
                    Color.Navy,         // color
                    (float)m_EffectAge, // cursor rotation
                    center,             // center of cursor
                    1.0f,               // don't scale
                    SpriteEffects.None, // no effect
                    0.0f);              // topmost layer
            }
        }
    }
}
```

The DrawPieces Method

The `DrawPieces` method renders every piece on the game grid.

```
// draw any existing game pieces
private void DrawPieces(SpriteBatch batch)
{
    Vector2 position = Vector2.Zero;
    for (int y = 0; y < GridHeight; y++) // for each row
    {
        position.Y = TopLeft.Y + y * 32;
        for (int x = 0; x < GridWidth; x++) // for each column
        {
            position.X = TopLeft.X + x * 32;
            Grid[x, y].Draw(batch, position);
        }
    }
}
```

The DrawHUD Method

The `DrawHUD` method renders the player scores, along with a visual indicator to denote the current player.

```
// score for player one and player two
NumberSprite m_ScoreOne = new NumberSprite();
NumberSprite m_ScoreTwo = new NumberSprite();
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 418


```
// extra pieces for HUD
GamePiece m_PieceOne = new GamePiece(Player.One);
GamePiece m_PieceTwo = new GamePiece(Player.Two);

// render the heads up display
private void DrawHUD(SpriteBatch batch)
{
    Vector2 position = Vector2.Zero;
    Vector2 center = new Vector2(16, 16);

    position.X =
        TopLeft.X +      // left of grid
        GridWidth * 32 + // right of grid
        2 +              // plus border
        32;              // plus some space
    position.Y =
        TopLeft.Y +      // top of grid
        GridWidth * 16 - // middle of grid
        32;              // minus one row (since HUD takes up two rows)

    // draw gear around player one's HUD piece?
    if (State == GameState.Playing && CurrentPlayer == Player.One)
    {
        position += center - Vector2.One;
        batch.Draw(
            Texture,            // cursor texture
            position,           // cursor x, y
            CursorRect,         // cursor source rect
            Color.White,        // color
            (float)m_EffectAge, // cursor rotation
            center,             // center of cursor
            1.0f,               // don't scale
            SpriteEffects.None, // no effect
            0.0f);              // topmost layer
        position -= center - Vector2.One;
    }
    // draw player one's HUD piece?
    batch.Draw(
        Texture,
        position,
        GamePiece.PieceRect,
        GamePiece.Colors[1]);
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 419


```
    // draw gear around player two's HUD piece?
    position.Y += 32;
    if (State == GameState.Playing && CurrentPlayer == Player.Two)
    {
        position += center;
        batch.Draw(
            Texture,             // cursor texture
            position,            // cursor x, y
            CursorRect,          // cursor source rect
            Color.White,         // color
            (float)m_EffectAge, // cursor rotation
            center,              // center of cursor
            1.0f,                // don't scale
            SpriteEffects.None, // no effect
            0.0f);               // topmost layer
        position -= center;
    }
    // draw player two's HUD piece?
    batch.Draw(
        Texture,
        position,
        GamePiece.PieceRect,
        GamePiece.Colors[2]);

    // draw scores
    position.Y -= 26;
    position.X += 32;
    m_ScoreOne.Draw(batch, position);
    position.Y += 32;
    m_ScoreTwo.Draw(batch, position);

    if (PlayerTypeRect != Rectangle.Empty)
    {
        position.X -= 16;
        position.Y += 32;
        batch.Draw(Texture, position, PlayerTypeRect, Color.White);
    }
}
```

The DrawProgress Method

The `DrawProgress` method renders a progress bar that represents the state of completion of the currently selected AI. This lets the human player (if any) know

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 420

that his CPU opponent is still thinking, and that the game is not "locked up." It's important to provide feedback to your players when you're performing long-running tasks.

```
// if CPU player is working, show progress on screen
private void DrawProgress(SpriteBatch batch)
{
    if (State == GameState.Playing)
    {
        // assume zero progress
        double progress = 0;

        // grab progress from current CPU player, if any
        ArtificialIntelligence ai = CurrentAI;
        if (ai != null)
        {
            progress = ai.Status;
        }

        // actually draw progress bar
        Rectangle rect = new Rectangle(
            (int)TopLeft.X,
            (int)TopLeft.Y + GridHeight * 32,
            (int)Math.Round(progress * (GridWidth * 32)),
            2);
        batch.Draw(Texture, rect, PixelRect, Color.White);
    }
}
```

The GamePiece

The `GamePiece` class holds only one important piece of data—its owner (to which a reference is stored in the `GamePiece.Owner` property). `GamePiece` objects are responsible for rendering themselves in the appropriate color, but don't really perform any other noteworthy tasks.

```
public struct GamePiece
{
    // simple copy constructor
    public GamePiece(Player owner)
    {
        Owner = owner;
    }
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 421


```
    // texture rectangle
    public static readonly Rectangle PieceRect =
       new Rectangle(0, 0, 32, 32);

    // color, based on owner
    public static readonly Color[] Colors = {
        Color.TransparentWhite,
        Color.Goldenrod,
        Color.OrangeRed,
    };

    // owner of this piece
    public Player Owner;

    // update this piece
    public void Update(double elapsed)
    {
    }

    // render this piece at the specified position
    public void Draw(SpriteBatch batch,Vector2 position)
    {
        if (Owner == Player.None) return;
        batch.Draw(
            GameBoard.Texture,
            position,
            PieceRect,
            Colors[(int)Owner]);
    }
}
```

NumberSprite

Rather than using a full-blown font rendering class, this class is a simple helper that draws numbers on the screen. We'll use this class to render our scores. `NumberSprite` exposes a public, long property called `Value`. This is the number that will be rendered. The `NumberSprite` class stores its texture and texture rectangle data in a static variable so that it's accessible from all instances of the class. This texture must have a special layout. The `NumberSprite` class assumes that there are 10 evenly spaced, single-digit numbers in the texture on a single row, in order from zero to nine. The `Draw()` method of this class renders each digit of `Value`, one by one.

```
class NumberSprite
{
    // texture is shared across all instances (static)
    private static Texture2D m_Texture;
    private static Rectangle[] TextureRects = new Rectangle[10];
    public static Texture2D Texture
    {
        get { return m_Texture; }
        set
        {
            // set texture
            m_Texture = value;

            // texture is 10 evenly spaced numbers
            int widthChar = Texture.Width / 10;
            int heightChar = Texture.Height;
            for (int i = 0; i < 10; i++)
            {
                TextureRects[i] = new Rectangle(
                    i * widthChar,
                    0,
                    widthChar,
                    heightChar );
            }
        }
    }

    // actually draw the number (using default White tint)
    public void Draw(SpriteBatch batch, Vector2 position)
    {
        Draw(batch, position, Color.White);
    }

    // actually draw the number
    public void Draw(SpriteBatch batch, Vector2 position, Color tint)
    {
        // draw the number, char by char, from cache
        for (int i = 0; i < m_ValueAsText.Length; i++)
        {
            int c = m_ValueAsText[i] -'0';
            batch.Draw(Texture, position, TextureRects[c], tint);
            position.X += TextureRects[c].Width;
        }
    }
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 423

Whenever the `Value` property is updated, the new value is compared to the existing value. If the value is changing, then we update the internal value (`m_Value`) and use the APIs in the .NET Framework to create the array of text characters that represent this number. Whenever we're ready to draw the number, we'll refer to this array of characters rather than the number. Each character can be converted to an index into our array of bitmaps by subtracting the '0' character literal. By caching the character array, we don't need to calculate it every time the number is drawn (every frame of the game). We only need to generate the character array when the value actually changes.

```
    // cache a copy of the last value to save some CPU cycles
    private char[] m_ValueAsText = "0".ToCharArray();
    private long m_Value = 0;
    public long Value
    {
        get { return m_Value; }
        set
        {
            if (m_Value != value)
            {
                m_Value = value;
                m_ValueAsText = value.ToString().ToCharArray();
            }
        }
    }
}
```

The User Interface

This game uses threads to keep the animations running smoothly, even when the processor is busy calculating the next move for a computer opponent. While the CPU is busy planning its next move, human input is blocked.

The HUD (Heads-Up Display)

The game displays a spinning gear in every cell of the grid that would be a valid move for the current player. The game allows for either a human or computer opponent as player two. If the current player is human, one of the gears is highlighted, representing the currently selected move. If the current player is a CPU opponent, no gears are highlighted and a simple progress meter is displayed at the bottom of the grid to let player one (always human) know how the AI is

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 424

Table 18.1 Player Input

Keyboard	Game Pad	Description
Left	Left	Highlight previous valid move
Right	Right	Highlight next valid move
Space	A	Make the currently highlighted move
Page Up	Left Shoulder	Lower CPU difficulty or allow human as player two
Page Down	Right Shoulder	Raise CPU difficulty or allow human as player two
Enter	Start	Restart the game

progressing. If the input for player two is being managed by AI, the level of difficulty of the CPU player is noted in the HUD as well. The HUD logic also indicates the current player by rendering a spinning gear next to their stats in the HUD, and it displays the current score for each player.

Player Input

The game supports both the keyboard and the game pad. If there are two human players, they can take turns on the keyboard or on a single controller. If there is a controller active on port two, then each player can use his own controller. As mentioned earlier, human input is blocked while the AI is processing. This keeps thread synchronization simple by preventing the creation of multiple, concurrent AI threads.

The list of buttons and their functions are shown in Table 18.1

There is a delay of a quarter of a second between each button press. This makes handling repeating button presses easier. Rather than tracking each button's pressed and released state from frame to frame, the delay serves as a global repeat rate. The biggest drawback to this method of processing player input is that it doesn't allow for button combinations. Considering the type of game that we're writing, that's not such a big issue, though. And the savings in code complexity far outweigh any potential loss in player input design options.

Artificial Intelligence

To make things a little more interesting, this game supports two players. The second player can be human, but the game also provides four CPU opponents that the player can challenge. These computer opponents are processed on a separate thread so that the game animations can continue uninterrupted.

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 425

The Base Class

The base class for the CPU opponent (ArtificialIntelligence) manages the creation of worker threads, handles thread synchronization tasks, and provides a single interface to the GameBoard class for all the CPU opponents that we will develop. While the AI is processing game data to decide which move to make, human inputs are ignored. This makes thread synchronization easier by ensuring that only one AI thread is active at any one time.

The GameBoard class is responsible for populating the list of valid moves whenever a new turn begins. The derived AIs are responsible for determining which move they would like to make from that list, and they're responsible for reporting their progress back to the GameBoard class (for use in the DrawHUD method) by updating their Status property as they churn through all possible moves. The Status property is a double value, where 0.0 represents 0% completion and 1.0 represents 100% completion. Generally speaking, the derived class can assume that it's running exclusively, since the base class is handling all the threading tasks.

```
public abstract class ArtificialIntelligence
{
    // a default constructor
    public ArtificialIntelligence()
    {
    }

    // constructor with depth initialization
    public ArtificialIntelligence(int depth)
    {
        Depth = depth;
    }

    // used by UI to render progress bar
    protected double m_Status = 0;
    public double Status
    {
        get { return m_Status; }
    }

    // used by GameBoard to know when the AI is done taking its turn
    protected bool m_Done = true;
    public bool Done
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 426


```
{
    get { return m_Done; }
}

// the move that the AI selected, as an index into
// GameBoard.ValidMoves[]
protected int m_Move = 0;
public int Move
{
    get { return m_Move; }
}

// number of levels to recurse when searching all possible board combos
protected int m_Depth = 4;
public int Depth
{
    get { return m_Depth; }
    set { m_Depth = value; }
}

// handy member to generate random values
protected Random m_rand = new Random();

// the public interface to other classes, kicks off threaded
// task, then returns to caller. caller polls AI.Done to see
// if threaded task is done
public void SelectMove(GameBoard board)
{
    m_Done = false;
    m_Move = 0;
    m_Status = 0;
    SelectMoveParams param = new SelectMoveParams();
    param.Board = board;
    param.Depth = Depth;
    Thread task = new Thread(SelectMoveTask);
    task.Start(param);
}

// parameters for the threaded task
public struct SelectMoveParams
{
    public int Depth;
    public GameBoard Board;
}
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 427


```
    // the method that is actually called when threading starts
    protected void SelectMoveTask(object obj)
    {
        SelectMoveParams param = (SelectMoveParams)obj;
        m_Move = SelectMoveRecursive(param.Board, param.Depth);
        m_Status = 1;
        m_Done = true;
    }

    // helper method to generate a simple heuristic for a given GameBoard
    // add 1 for each piece owned by Player.One, subtract 1 for each piece
    // owned by Player.Two. Larger sums favor Player.One, smaller sums
    // favor Player.Two.
    protected int Evaluate(GameBoard board)
    {
        return board.Score(Player.One) - board.Score(Player.Two);
    }

    // this method must be implemented by any actual AIs that derive
    // from this base class. the threaded method calls this method,
    // which only exists in derived classes. that way, this base
    // class can handle the nitty-gritty threading and synchronization
    // tasks, and leave the actual AI processing to the derived classes.
    protected abstract int SelectMoveRecursive(GameBoard board, int depth);
}
```

RandomAI

This is about as simple a CPU opponent as we could create. The "easy" opponent (`RandomAI`) randomly selects a single move for the list of valid moves. The only AI that would be easier than this would be one that selects the first valid move every time, but that would be really boring. Every now and then, this CPU opponent may get lucky and select a great move. But, over time, this AI should be fairly easy to beat.

```
class RandomAI : ArtificialIntelligence
{
    // constructor with depth initializer
    public RandomAI(int depth) : base(depth) { }

    // as simple as AI gets -- random selection
    protected override int SelectMoveRecursive(GameBoard board, int depth)
    {
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 428


```
        // randomly select a move from the list of valid moves
        return m_rand.Next(board.ValidMoves.Count);
    }
}
```

EasyAI

Another strategy that we can use is to pick the best move for the current turn. We can make each valid move on a copy of the current game board and see which one improves our score the most. While this makes for a more challenging opponent than the random AI, it's a very short-sighted strategy. In this game, there are many times when a move that gains you many pieces in a single turn will lead to a move where your opponent can reclaim even more pieces on the next turn.

```
class EasyAI: ArtificialIntelligence
{
    // constructor with depth initializer
    public EasyAI(int depth) : base(depth) { }

    // called by base class from separate thread
    protected override int SelectMoveRecursive(GameBoard board, int depth)
    {
        // assume first move is the best
        int move = 0;

        // best score, based on player
        if (board.CurrentPlayer == Player.One)
        {
            move = FindBestMove(board, true, int.MinValue);
        }
        else if (board.CurrentPlayer == Player.Two)
        {
            move = FindBestMove(board, false, int.MaxValue);
        }

        // return selected move
        return move;
    }
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 429


```
        protected int FindBestMove(GameBoard board, bool isPlayerOne, int score)
        {
            // assume first move is the best
            int move = 0;

            // scan valid moves, looking for best score
            for (int index = 0; index < board.ValidMoves.Count; index++)
            {
                // create copy of current board to play with
                GameBoard board2 = new GameBoard(board);

                // make the next valid move
                board2.MakeMove(index);

                // what's the score?
                int val = Evaluate(board2);

                // best score for player one is positive,
                // best score for player two is negative
                if (isPlayerOne)
                {
                    if (val > score)
                    {
                        score = val;
                        move = index;
                    }
                }
                else
                {
                    if (val < score)
                    {
                        score = val;
                        move = index;
                    }
                }
            }

            // report findings to the caller
            return move;
        }
    }
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 430

MinMaxAI

Reversi falls into a category of games known as "zero sum games with perfect information." That means that when one player wins, the other loses (a zero sum game). It also means that you can inspect the game board at any time during the game and know who's winning at that instant and know all of the moves that are legal for the current turn (perfect information). There is no chance element to this game. Other games in this category are checkers, chess, connect four, and tic-tac-toe. Games like these are ideal candidates for creating virtual opponents using an algorithm known as Min/Max.

The Min/Max algorithm attempts to minimize its opponent's score while maximizing its own. It does this by recursively examining all the possible moves, counter moves, counter-counter moves, etc. This is similar to what the grand-masters of checkers and chess do. To win at this type of game against a veteran player, you need to think several moves ahead of the current turn. "If I move here, and he moves there, I can move here and win!"

Where a human player relies on experience, skill, and instinct, the computer player uses simple brute force—playing out every possible scenario to see how it turns out. Even though computers are great at processing large sets of data like this, and they're able to consider more moves faster than any human can, there are limits. As you examine these potential moves further into the future, the number of combinations grows exponentially.

If we assume that there are an average of eight legal moves in a turn (and eight legal counter moves, and eight legal counter-counter moves, . . .), then you would have "8 to the power of the recursion depth" `GameBoard` objects to create, manipulate, and examine. So if we wanted to examine 12 moves into the future, we'd have to look at just under 69 billion game boards and decide which move (from the initial eight valid moves) would likely lead to the best end result.

Of course, most games last longer than 12 moves. But we can't just keep looking further ahead. Just looking one more move ahead in our hypothetical scenario would require examining another 480 billion game boards. Your PC just isn't big enough or fast enough to process that kind of data before the player gets bored and shuts your game off. Clearly, we'll need to make a decision long before we see how the game ends.

Since we're selecting our move before we know for certain that it will lead to our victory, there's always a chance that we're choosing a short-term gain that will ultimately lead to our opponent's victory. We can't just blindly select the move

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 431

that yields the highest score for us. That's too short-sighted a strategy. The way that the Min/Max algorithm handles this problem is by selecting the move that leads to the highest score for us, while simultaneously limiting the gains made by our opponent. It's not perfect, but it's a solid strategy. The more future moves that you can examine, the better your decisions for the current move will be.

Clearly you can't dictate what move your opponent will make, but you can safely assume that he will make the best possible move that he can. If he doesn't, then you'll have an even better set of moves to pick from on the next turn. With each recursive call, the AI will assume the role of the new player and pick the best move for that player every time.

Pseudo code for the Min/Max algorithm follows:

```
Begin Max Method
    If RecursionDepth is Zero Then
        Return Score
    End If
    For Each Valid Move
        Make Move
        NewScore = Min( RecursionDepth - 1)
        If NewScore > Score Then
            Score = NewScore
        End If
        Undo Move
    End For
    Return Score
End Max Method

Begin Min Method
    If RecursionDepth is Zero Then
        Return Score
    End If
    For Each Valid Move
        Make Move
        NewScore = Max( RecursionDepth - 1)
        If NewScore < Score Then
            Score = NewScore
        End If
        Undo Move
    End For
    Return Score
    End Min Method
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 432

The complete listing for this class can be found on the CD-ROM that came with the book.

AlphaBetaAI

The Min/Max algorithm does a wonderful job, and the deeper we allow it to search, the better it is at picking great moves. We can optimize our implementation as much as we like, but the sheer number of game board combinations that we have to examine dwarfs any potential performance boosts that we may come up with. To really speed things up, we need to limit the amount of data that the algorithm needs to examine. Well, it just so happens that there is a way to safely ignore many of those game board combinations.

The Min and Max methods exchange the same dialog, over and over—"If I move here, where will you move?" Max will always pick the best move for player one and Min will always pick the best move for player two. So, what is "the best move"? The best move is the one that presents the worst options for your opponent. If you choose a move for which your opponent has a brilliant counter-move, there's a good chance that he'll take it. So the strategy that Min/Max uses is to choose a move such that the best counter-move is as undesirable as possible.

For each possible move, Max will ask Min what his best counter move would be. Max then selects his move based on the worst of the counter-moves that Min reported. Understanding this exchange is essential in limiting the number of game boards that we need to process.

If Max is looking for Min's worst counter-move, then Min can stop evaluating game boards as soon as he realizes that the current list of valid counter-moves contains a move that's better than a counter-move that he's already reported to Max. Why? Because Min will chose a counter-move that's at least as good as the counter-move he's currently evaluating (that is, after all, his goal). There's no reason for Min to find the best counter-move if Max is sure to throw the results out anyway. That, in a nutshell, is the logic behind the Alpha/Beta algorithm.

The following is a bit of pseudo code that demonstrates the basic Alpha/Beta algorithm. Notice how similar it is to the Min/Max algorithm.

```
Begin Max Method
    If RecursionDepth is Zero Then
        Return Score
```

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 433


```
    End If
    For Each Valid Move
        Make Move
        NewScore = Min( RecursionDepth - 1)
        If NewScore >= Beta Then
            Return Beta
        End If
        If NewScore > Alpha Then
            Alpha = NewScore
        End If
        Undo Move
    End For
    Return Alpha
End Max Method

Begin Min Method
    If RecursionDepth is Zero Then
        Return Score
    End If
    For Each Valid Move
        Make Move
        NewScore = Max( RecursionDepth - 1)
        If NewScore <= Alpha Then
            Return Alpha
        End If
        If NewScore < Beta Then
            Beta = NewScore
        End If
        Undo Move
    End For
    Return Beta
End Min Method
```

The complete listing for this class can be found on the CD-ROM that came with the book. While Alpha/Beta produces roughly the same results as Min/Max for a given recursion depth, it does so much faster. That means that you could increase the recursion depth for Alpha/Beta until it takes as long to run as the unmodified Min/Max algorithm and make better decisions with the same amount of processing time. For this chapter's example, I kept the recursion depths for these two AIs the same so that you get comparable decisions to the CPU Hard opponent (which uses a vanilla Min/Max implementation), in much less time.

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 434

Summary

In this chapter you learned how to implement simple computer opponents to enhance single-player game play. You saw how adding simple animations and other user interface niceties can make plain game designs more appealing. You also learned how to handle player input by adapting to the available input hardware. This chapter even included a little threading code to keep the animations running while the AI was busy deciding what move to make.

The example code for this chapter is by no means fully optimized. It's just an illustration of how you might go about implementing AI in your game. For more information on threading, read Chapter 25, "Threading in XNA."

Review Questions

Think about what you've read in this chapter (and the included source code) to answer the following questions.

1. How does the game ensure that players can only make legal moves?
2. How does the game keep the animations running smoothly, even when a computer opponent is busy processing game board data?
3. What are the differences between the three AI implementations?
4. Why does the game disable player input while the artificial intelligence is processing?
5. The implementation for the `RandomAI` class is very simple. Why do you think it's threaded just like its CPU-intensive siblings?

Exercises

EXERCISE 1. Modify this example so that the player can change the recursion depth of the selected AI to make the game easier or harder to suit their taste.

EXERCISE 2. As written, the Min/Max routine will always make the same move for any given game board. In many cases, there may be more than one move that will yield the same end result. Since Min/Max is evaluating every possible game board anyway, keep track of all moves that yield the same result and randomly pick a move from this list of "best" moves. That little bit of variety will make the computer opponent more interesting.

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 435

EXERCISE 3. Implement an "undo" feature for this game to allow the human player to reconsider his last move. Since every game board state is completely independent of every other game board state, this should be as easy as remembering the state of the board before the last human turn.

EXERCISE 4. Min/Max is by far the most popular implementation for AI to tackle this type of game. Come up with an original, novel AI subclass. Don't worry if it's not as strong a player as the existing AIs—just make something original. Experiment with more complex heuristics than simply comparing scores. For example, you might weigh piece placement so that grid cells on the edge of the board are more valuable than grid cells in the center (assumes that edge pieces are less likely to be recaptured). Be creative.

d:/Temp_Dailyproduction/Game_Dev_163020/z_production/z_3B2/Game_-
Dev_1598633686_ch18.3d, 8/14/7, 18:56, page: 436