MonoGame World Editor + GUI guide

Contents

GAME LOOP AND RENDERING

GraphicsDeviceManager graphics - represents the graphics device - video card.

SpriteBatch spriteBatch = helper objects for scheduling the drawing order of sprites and text. spriteBatch = new SpriteBatch(GraphicsDevice); use spritebatch.begin() and end() functions to perform rendering. much faster if all of the same images are rendered at once and switch only happens when next group of images is to be drawn.

RenderTarget2D - these objects are used as a canvas for drawing objects, lines, text or anything else. Default RenderTarget2D is the screen itself. GraphicsDevice.SetRenderTarget(target name here) function is used to switch the current drawing canvas. To clear the canvas use clear() function then with spritebatch begin() and end() functions the render target can be painted. Multiple layers can be achieved this way as well as separation of shader effects.

To create effect on current render target in between spritebatch being and end you need to use a proper shader effect file and set any relevant parameters where applicable,e.g. fx_ui_masking_shader.Parameters["maskTexture"].SetValue(world_ui_mask_layer); then apply the effect on the currently drawn objects like this: fx_ui_masking_shader.CurrentTechnique.Passes[0].Apply();

MAIN GAME LOOP COMPONENTS

INITIALIZE()

Assign render targets, graphic options, pixel textures.

Need to take note: Initialize base function also call the load content function!

LOADCONTENT()

Load effects, textures, fonts, user interface from the xml files, worlds from the xml files, deserialize UI positions and colors.

UNLOADCONTENT()

Serialize gui and worlds, release all Content textures from memory.

UPDATE

refreshes variables based on user activity

Current update order:

- get **keyboard and mouse** states before doing anything else to have previous and current states available at the same time
- set mouse **wheel** value
- update game **clock** milliseconds
- **calculate fps** value
- update **editor** - slider values, switch buttons , text areas, editor modes etc.
- update **Engine** object's text engine

- accept keyboard **controls** - key presses and perform any appropriate actions signalled by them
- save keyboard and mouse to **previous states**
- generate **statistics**

DRAW

Current drawing order for the project is:

- UI
    - **world_ui_mask_layer** - uses public override void draw_masking_sprite from the base ui element class. overridden in,for example, in progress bar element, a special masking texture with black and white colors hides all parts of the background that have a black pixel corresponding to the same pixel on the background.
    - **world_ui_layer** - draw all the static UI elements (not context pop-up elements. draws all the visible containers and elements inside them.
    - **world_ui_mask_temp** - a render target that combines the actual elements with their masks erasing all the masked portions of the element. Because only the UI render targets are involved - none of the world features are affected. world_ui_layer is then drawn to this temp target.
    - **world_ui_mask_temp** then goes through seconds phase of drawing - where post processing, context containers and tooltips are drawn (to prevent contexts and tooltips which should appear on the topmost layer of scene from being masked by masking layer, since tooltips and contexts can overlap static elements).post processing includes texts displayed inside textarea elements, pink outline for containers if GUI is in element movement mode, tooltips and borders. For example, a border around text input field under system chat. **public override void draw_post_processing** function is used. Important note: since every sector has its own content type, user must specify which sector content should be surrounded by a border. copy + paste from one element to another will lead to skipped rendering steps if that element doesn't have the same content type.
- After all UI layers are created, main game scene is drawn.
    - **lighting - fx_lightmap_shader** parameter is being set with ambient light (very dark if midnight , almost full ambient light if midday)
    - **world_light_buffer** - tearget that gets painted with all light spheres (circles)
    - **world_tile_buffer** gets painted with all the tiles in the current active world
    - switch to main screen render target - add **world_light_buffer** to lights texture parameter of the **fx_lightmap_shader**, then GraphicsDevice.Clear() is used to paint the background of the world (sky) with color based on time of day in-game. If lighting is on - **fx_lightmap_shader** combines the ambient light with point light circles to simulate darkness or colored lighting over certain areas (for visible areas of the world). Combine the light map with **world_tile_buffer** to create shadows and lights on the world.
    - continue drawing to main screen using **draw_world_geometry(), world_draw_point_light_sources(), world_draw_water_generators()** functions. geometry consists of border lines and crosshairs.
    - finish up by drawing world_ui_mask_temp containing entire user interface in its current state to the screen.
- **Engine draw()** - final scene rendering
    - draw current time in the right top corner using **engine.xna_draw_outlined_text()**
    - draw mouse hand and statistics text in the appropriate text area.

SHADERS

Graphic effects used to manipulate scenes on per-pixel level through graphics card.

fx_lightmap_shader explanation (**written in HLSL shader language**):

```
// texture sampler - world map render target loads here

sampler s0 : register(s0);

Texture2D lightsTexture;                                        // load render target here

sampler lightSampler = sampler_state{                           // light mask texture - point lights render target sampled here

        Texture = <lightsTexture>;

};



float   intensity;                                              // [parameter] - intensity of ambient light (lower is darker) based on World clock

//float   radius = 100;                 // testing radius of circle (reach distance of world illuminating object)

//float   light_coords[2000];                                   // space for 1000 light coordinates - testing iteration through coordinates, create corresponding
array in game class

//-------------------------------------------------------------

// shader function

float4 PixelShaderFunction(float4 pos : SV_POSITION, float4 color1 : COLOR0, float2 coords : TEXCOORD0) : SV_TARGET0

{

        float4 color = tex2D(s0, coords);       // world scene texture: tiles

        float4 lightColor = tex2D(lightSampler, coords);  // color information of the lights buffer


        // limit light intensity based on world ambient light value

        lightColor.a *= 1.0f - intensity;


        // anything that doesn't have point lights is unaffected

        if (lightColor.a == 0.0f)

                return color*intensity;


        // add color to lighting

        if (color.a != 0.0f)

                color.rgb += (lightColor.rgb*0.65f); // last floating point number adjusts the power of color component


        // finish up

        if (intensity + lightColor.a <= 0.9f)

                return color*(intensity + lightColor.a);

        else
```

```
                return color*0.9f;
}
// technique (no need to change this function)
technique Technique1
{
        pass Pass1
        {
                PixelShader = compile ps_4_0_level_9_3 PixelShaderFunction();
        }
}
```

sampler s0 = storage for the render target to which the effect is applied.

lightstexture = a parameter later sampled into lightSampler, so **s0** contains **world_tile_buffer (world)** and **lightSampler** contains **world_light_buffer (ambient lights + colored spotlights)**.

> *previous version of the shader used array of colors to store lights, however, due to limitations of GPU for loops processing these lights could only handle 3-4 lights at a time, compaired to limitless lights created with a mask.*

intensity is the parameter specifying light brightness (low values are darker).

float4 color = tex2D(s0, coords); - contains all pixel color data of the world tiles.

float4 lightColor = tex2D(lightSampler, coords); - contains all pixel color data of the light mask.

lightColor.a *= 1.0f - intensity; - intensity of the lights is high when intensity of the ambient light is low(night) and low when ambient is high (day).

if (lightColor.a == 0.0f) return color*intensity; - these areas are completely unaffected by the point lights, so only the ambient light is applied to the world.

color.rgb += (lightColor.rgb*0.65f); - last floating point number adjusts the power of color component - here colors of the world are additively blended with the color of the lighting to create the effect of lighting unique and true for each surface

return color*(intensity + lightColor.a); - finish up by making each pixel brightness capped at 0.9 or ambient brightness + color light brightness.

Other helper functions of the Game1 class

- `serialize_map_data`
- `Texture2D createCircle`
- `Texture2D createHalfCircle`
- `Texture2D create_colored_rectangle`
- `Texture2D create_colored_hollow_rectangle`
- `update_resolution`
- `make_fullscreen`

WORLD

(Tile,TileMap,DataTypes,WorldCollection)

This is a series of classes created specifically for the purpose of testing the editor, GUI and other framework extending functionalities. **It is not necessary to know the structure of these entities for future projects.**

IMPORTANT COMPONENTS:

- WorldCollection - a list which defines what filename contains which world (WorldStruct).Changes active world and is responsible for deserialization through load_tiles function.
- MyDataTypes.MapData - a class that is used to load the tile definitions from the xml files and build the World. Has name and position saved for every tile. (There are also classes responsible for UI deserialization is the namespace)
- Tile static class containing List<tile_struct>.Defines all possible tiles for the World. Used to find tiles by name or id
- [,]tile_map - multidimensional array (not jagged) containing tile_map structures. Each such structure contains among other things unique tile id, variant, corner information. Based on the type a different spritesheet is used.
- List<PointLight> - a collection of all the light sources (PointLight) in the given world. Lights are defined by their unique colors, poositions in relation to map origin, cell address (one light per cell), reach radius, intensity, source sprite and light_sphere texture generated for the light mapping shader to actually display the effect of the light.
- List<WaterGenerator> - contains elements responsible for creating new water in a given tile. Similar to light, except it has no texture for effects.
- Rectangle[] corner_src - helps with drawing missing inner corner borders for a tile surrounded by other tiles, e.g. if there is a tile on top and left the top left corner texture is added to the center tile.

DRAW SEQUENCE

- define an area of the world visible inside current viewport
- for each tile type - draw all the tiles of this type before going to the next one to greatly optimize the draw time
- determine what version of the tile is needed - based on the neighboring tiles
- draw water

UPDATE

- calculate sky color
- recalculate tile versions based on any new/deleted cells only (updated_cells list)
- generate new water then calculate water changes

EDITOR

Explanation of how the editor works, what is the flow of each operation, what is required from the user/system to initiate, how to add features or debug processes.

Editor is responsible for shaping the World object by manipulating tiles that belong to it.

MODES:

- add mode - creates new tiles in single/radius brush configuration, line, rectangle, hollow rectangle. Creates water generators in water tool mode.
- delete mode - opposite of the add mode with the same tools
- selection mode - creates selection matrix that holds cells, lights and water generators. Anything inside the selection can be deleted, cells can switch their type, cells can be created (same effect as add mode's rectangle tool. *lc command changes the light color of any lights in the selection.
- lights mode - creates a point light in selected cell. only one light per cell is allowed
- water mode - creates a water generation at mouse pointer

Submodes have their own menu and can be enabled for add/delete mode. This menu is only visible for those modes.

Water has been moved to its own mode.

color_theme structure - color and transparency of each theme version is stored with uniqe string id. A string surrogate of the color object is created and used to serialize and deserialize.

load content() - builds **CONTAINER_EDITOR_TILE_CHANGER** based on the available tile types. This tool allows changing current active tile type either from contex menu or dedicated UI element.Builds interface theme context menu. assigns newly created subcontexts to correct buttons. loads slider values tying them to in-game variables. generates textures for UI element backgrounds. assigns custom background textures to buttons.enables scrollbar for tile selector element.assign a textarea to main text input.

load slider values - based on the actions enumeration find a slider element that represents the action and assigns a default min max values as well as precision (only for float) to it. Any switch buttons are initialized with a true/false value. progress bars and circles are also initialized.

draw - borders, indicators of add/delete mode, some text, editor instructions, tile previews.

update - GUI elements are updated first, then based on the mode - selection, line tool drivers are either active or inactive.

EDITOR_COMMAND() - ties user interface and user keyboard presses to editor actions. Each distinct action has it's own enum variable for easy control flow. It is also responsible for hover/ no hover sections and command section.

Preparation section:is UI is unlocked (no mouse drag on slider and no ui movement mode) - calcualte command based on hover, this will later be used in action switch.

Flow:

1. Execute command - no hover : includes functionality for key shortcuts as well as mouse.
   Note: different actions are taken if there is a hover over UI element.
2. Action switch - if a command together with hover returned an actiuon to be executed (calculated in preparation section): in move mode the only action available is a movement lock on hover and mouse release of element that has unlock_ui_move action. Outside of movement mode: actions depend on editor being active, hover status and editor mode.
    a. no hover, mode dependent (by mode) : here based on the mode you can paint new tiles, delete tiles, manage lights, water etc. based on variety of available tools
        i. add mode
        ii. delete mode
        iii. select mode
        iv. light mode
    b. no hover, mode independent (by command)
        i. left click command - unfocus input
        ii. ctrl+click command - selection matrix expansion
    c. hover, mode independent(actions that are available for any editor mode)
        i. clear all
        ii. fill all
        iii. editor mode switch (4 variants)
        iv. multiple resolution changes
        v. center world
        vi. summon subcontext menu
        vii. update slider value
        viii. update colors/themes
        ix. brush size change
        x. randomzie sliders
        xi. main context menu summon
        xii. switch the bool button
        xiii. change current active cell type
        xiv. hide contexts
        xv. make element invisible
        xvi. focus input
        xvii. toggle system chat visibility

UI/Editor Action	Mouse Keyboard commands

public enum actions
{
editor_mode_switch_add = 0,
editor_mode_switch_delete,
editor_mode_switch_select,
editor_mode_switch_lights,
game_exit,
overall_context,
toggle_subcontext,
hide_menu,
clear_all,
fill_all,
update_slider_grid_transparency, // editor grid
update_slider_grid_color_red,
update_slider_grid_color_green,
update_slider_grid_color_blue,
update_selection_transparency, // selection matrix
update_selection_color_red,
update_selection_color_green,
update_selection_color_blue,
update_interface_transparency, // User Interface
update_interface_color_red,
update_interface_color_green,
update_interface_color_blue,
hide_this_container,
none, // this action will have nothing assigned to it but it will not be null
update_brush_size,
current_container_random_sliders, // sets ALL sliders in current context randomly
option_value_switch, // for value trackers - switch value
change_cell_design, // switch current active editor cell
color_preview, // action that doesn't change anything - simply identifies element
focus_input, // for text input - will make current hovered text_input element receive input until cancellation
switch_theme,
unlock_ui_move, // unlocks specific ui movement mode (reposition ui elements on screen)
resolution_fullscreen, // changes screen resolution to fullscreen mode
resolution_fullscreen_reverse,
resolution_1920_1080,
resolution_1440_900,
resolution_1366_768,
resolution_1280_800,
resolution_1024_576,
go_to_world_origin,
toggle_system_chat
};
	public enum command
	{
	left_click,
	left_hold,
	left_release,
	right_click,
	right_hold,
	right_release,
	alt_c,
	alt_e,
	alt_q,
	alt_1,
	alt_2,
	alt_3,
	alt_4,
	mouse_scroll_up,
	mouse_scroll_down,
	enter_key,
	delete_key,
	insert_key,
	destroy_water_gen,
	destroy_lights,
	ctrl_plus_click
	};

confirm action - protection from immediate action - for things such fill/clear entire world a confirmation is needed in case button was pressed accidentally.

accept_input() - if there is a focused input field - this function adds a new character with delay if the same or no delay if different.

line tool driver - based on the starting cell and direction of the mouse drag - create a horizontal (left to right or right to left) or vertical (top to bottom, bottom to top) line of new cells. When mouse is released all cells in that line are filled/deleted based on mode.

selection driver - builds a rectangle of selected cells based on start and end cell.

- building selection matrices of objects (lights and water generators(wg)) - for world objects such as lights and water generators a separate List is needed. Each existing light/wg is checked and if its cell address exists in selection matrix it is added to the list.Then based on user activity these lights/wg can be manipulated.

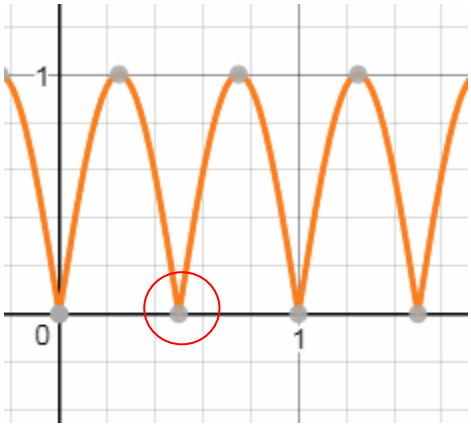
ENGINE

Support functions such as (and what processes they are used in):

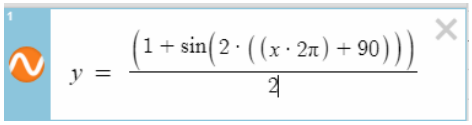
- move_camera - adjust visible portion fo the world. middle mouse button enables free dragging movement.
- is_key_char - input character is alphabetic
- is_key_digit - input character is number
- process_key - for input of text - capitalizes input, switches version of the character based on shift key.
- get_fps
- number_to_KMB - format number like 1245 to 1.2K etc.
- are_vectors_equal - compares two Vector2 objects
- generate_int_range - random number
- get_percentage_of_range
- generate_float_range - random floating number
- fade_up - **important** - increases the floating point value from some start value, after a delay and for a certain period of time until the max value.
- fade_down - **important** - same as above only opposite direction. goes own to 0 over time
- fade_sine_wave_uneven - **important** - calculate a remainder of current ms % duration divided by duration and multiplied by 360 degree angle to get a value between 0 and 360 degrees. Then find a sine of that angle - value between 0 and 1.

```
duration *= 2; // multiply by 2 to compensate for sine wave negative half (converted by Abs)

    float val = sine_wave(((float)((int)current_game_millisecond % (int)duration) / duration) * (Math.PI * 2));

    // flip negative half of the sine wave,

    // effectively generating two cycles in one duration,

    // therefore to keep desired duration of 1 cycle - multiply duration value by 2

return Math.Abs(val);
```

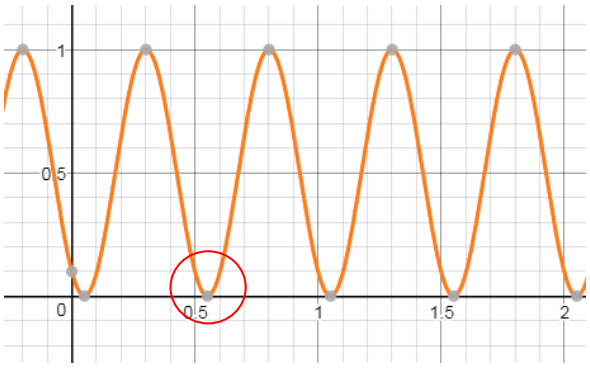
Equation 1 - update number over time



- fade_sine_wave_smooth - **important** - same as previous except the wave is much smoother because the function is brought up to eliminate negative half flipped by Abs() in previous version. notice the sharp angle in the first function and the smooth angles in this one.



Equation 2 - update number over time with smooth fade



- sine_wave - **important** - used to calculate sine of an angle.helper function for fade_sine family of functions.
- negative_color , adjusted_color, inverted_color - manipulate colors for contrast purposes (UI visibility)
- get_color_saturation, RGB2HSL, HSL2RGB - color system conversions
- rectangle_to_vector
- max_of_three
- min_of_three
- vector_centered
- get_texture_center
- calculate_orientation , line_intersection - collision of a line with another line
- serialize_data<T> - save current state of World or UI
- deserialize_data<T> - recreate World and UI since last program run
- rectangle_to_delimited_string - used to serialize rectangles in xml via surrogate string, since objects don't convert to xml
- delimited_string_to_rectangle - deserialize
- reverse string and number - both versions in one line
- isPowerofTwo - find if number is a power of two
- process_key() determines what character (and it's version) should be added to current focused input
- struct Int2 - analog of Vector2 but for integers

xna/monogame draw functions (**important**):

- xna_draw - standard - draw a sprite/texture
- xna_draw_rectangle_outline - draws 4 lines on the edges of rectangle
- xna_draw_rectangle - draws a rectangle shape with or without outline. **create_UI_backgrounds()** in **GraphicInterface** automatically creates the same type of background rectangle unless a custom background exists.
- xna_draw_text - uses **DrawString** function of the spritebatch to draw text on-screen
- xna_draw_text_crop_ui - crop the texture containing entire input to show the relevant portion contained inside text input
- xna_draw_outlined_text - draw text 5 times, first background shadow (the outline) by shifting text 1 pixel to top, bottom, left and right. Finish up by drawing original text in the original position.
- xna_draw_outlined_text_crop_ui - same as above but crops the length of the texture to only show the relevant parts

TEXT ENGINE

UPDATES:

1. {} construction in the text now groups multiple words together so the can all be colored the same. before each separate word had to be colored separately.
2. * keycharacter now identifies an action that is invoked by typing a command in system chat. E.g. *lc <parameter> will change the light color of selected lights in editor mode.

TEXT ENGINE STEP BY STEP PROCESS:

(creating text elements out of a string, replacing variables, performing actions, coloring words/groups of words)

Variables:

decode_mode - none, variable(~), action(*), color([])

message_element contains **text_element**, each text-element contains a single word and a color, message_element contains integer index of which word should start the next line.

Queue<message_element> messages contains message_element which contains List<text_element>

CREATING A MESSAGE ELEMENT

public message_element create_message_element(Engine e, string encoded_string)

1. variables
    a. List<text_element> grouped - a temporary list for text elements that should all be colored the same.
    b. message_element result - final message element created from the base coded string
    c. text_element temp_text - current text element being decoded - each one is added to message_element result.
    d. decode_mode dm (none, variable, color, action) - none = simple word, variable = construct variable name and replace it with some string, color = previously collected text element is colored with [] tag defined color. if there is a group active - color all the text elements in it as well. action = collect action name and perform action associated with it.
2. process
    a. add a space at the end of the string, since space is the condition of the text ending.
    b. replace variables with text, e.g. ~fps gets replace with a numeric value of current fps
    c. scan updated string character by character and decode colors, actions and groups
        i. if current character is not [,], {, }, * or space - add it to a word currently being built.
        ii. [character - switch mode to color - assign temp string as text of temp_text element
        iii.] character - complete the color mode - temp string now holds either a valid color name,e.g. red,green, etc. or a string of nnn,nnn,nnn format for rgb color. if not the fallback color is white. temp_text text is not changed in any way at this point. Color is assigned to current encoder. Since proper syntax of the command puts [] after group {} - all grouped messages are colored at this point, then added to the result.
        iv. { character - beginning of the group of messages, grouping flag is turned on, now every text element is added to the group instead of result until...
        v. } character - ends the grouping mode - followed by color tags
        vi. * character - switches decode mode to action

vii. space character - concludes most text elements based on :
    1. action mode: perform action and add response text to the text_element then add to result.
    2. no action mode: none mode = add a simple word to result, group = color element and add each one to result.

PERFORM_ACTION() - based on the action name - do something with the editor, engine, or UI.

*gl action - generates up to a 1000 point lights. Lights are created in the engine and only ordered by this command. each frame 1 light is created, so that rendering is not stopped for the duration of light creation. each new light occupies a unique location.

REPLACE VARIABLES() - substitute variable keyword with a string representing the value of that variable using decode_variable()

BREAKING MESSAGES INTO LINES

calculate_lines() function determines how many lines are in a message but doesn't actually generate the lines.

convert_to_ordered_lines() function - generates line breaks based on the width or presence of "/nl" text element.

message is broken with word break - one text element becomes two based on available area width. "/nl" keyword can be used to generate lines at will.

if (t.get_text().Equals("/nl")) - automatically break the line

if max width is exceeded - find the break point using:

while (engine_font.MeasureString(splitmessage.get_text()).X > (max_length - current_length))

while loop is used in case one text segment is so large it occupies many lines

split_message() is used to calculate proper breakpoint. creates 2 text_elements. 1st element is created by going back from the end of the string until remaining string fits in the available remaining space on the line. 2nd element is what remains of the string.

DRAWING IN RECTANGLE

public void textengine_draw(Engine engine, bool top_to_bottom = false)

top_to_bottom - flag determines vertical order of messages and starting point.

small_line_list contains all the text_element but in groups which represent a line of messages. this list is reversed, so that the newest messages are displayed first (because they are added to the list last). small_line_list is a subset of line_list but limited to the number of max lines allowed.

GRAPHIC USER INTERFACE

(Container,UIElementBase,UIElements,SectorStructs)

ADDING NEW BUTTON/ELEMENT:

add container in the ui_containers.xml file, then add an element with the newly added container's ID as parent. add element in the ui_elements.xml file. Add a new action type to the actions enum in enums.cs if necessary, then perform the action based on some user input (likely done in the editor_command). In editor.cs find the editor_command() function and add a case to switch statement. Inside the case do the appropriate work that should happen after the command is issued, for example, toggle visibility of the chat.

Default values for the state buttons are stored in load_initial_slider_values() inside editor.cs

Editor > loadContent() - change colors of the circle charts, set input target for the text area, load custom textures for the ui elements.

EXAMPLE

Adding a new context menu:

- create a container (with subcontext type)
- add a new element in the ui_elements.xml -1 in y parameter enables automatic sorting
- if a new action type - add to enum
- in load_initial_slider_values() - add any boundaries and current values to slider, or load custom background
- new action should be added to editor_command() switch in a proper section based on hover dependency
- updates can also be performed immediately outside of main update()

ENABLING THE ACTIVE INDICATOR

In GraphcInterface - use activate() function

ADDING NEW ELEMENT DEFINITION

Contains required functionality:

- base class UIElementBase -- all basic UI element features
- create_sectors() - create zones inside the rectangle of the element. horizontal zones filled with vertical zones each.
    - to create two or more rows (horizontal zones) - replace bounds.Height with properly calculated value
- draw - for each sector zone draw its content, such as text, icon , slider, etc. Define the way those are drawn uniquely for eac helement if needed.
- draw masking - draw a masking sprite - this will hide the area of the element making any kind of complex shape.
- draw post processing - add borders and other overlays onto the element

Once the new Class is created - add a new definition to the enumeration called "type"

Engine.load_user_interface() loads the entire user interface using two xml files: ui_containers.xml and ui_elements.xml. To make sure that this new element is recognized - deserialized ContainerData objects should create all the containers first. Then ElementData ui_type is converted to enumeration called "type". Based on that a proper UI element class object is created with available data and added to container. The internal design of each element is then drawn based on the sector zones.

```
// BASIC UI ELEMENT TEMPLATE - copy this to start a new element creation (all functions that require override are defined)
class ElementNameHere : UIElementBase
{
   public ElementNameHere(string id, Container parent, type f, actions? c, confirm safety, Rectangle dimension, Texture2D icon, String label,
String tooltip)
      : base(id, parent, f, c, safety, dimension, icon, label, tooltip)
   {
      create_sectors();
   }

   public new void create_sectors()
   {
      horizontal_sector temp_cb = new horizontal_sector(0, bounds.Height);    // create default

      temp_cb.add_vertical(new vertical_sector(0, bounds.Width, sector_content.circular_progress)); // circular progress chart will be fully
contained in 1 sector

      zones.Add(temp_cb);
   }

public override void draw(Engine engine, Color color, float transparency = -1f)
      {
         base.draw(engine, color, transparency);
         // element specific functionality
         foreach (horizontal_sector h in zones)
         {
            foreach (vertical_sector v in h.get_verticals())
            {  // rendering sequence - v contains x_start and x_end of the rectangle but y_start and y_end are in the "h"
               // calculate current vertical sector rectangle in on-screen coordinates based on "h" and "v" sectors and Unit origin(x,y)
               Rectangle draw_zone = new Rectangle(v.get_xs() + (int)get_origin().X, h.get_ys() + (int)get_origin().Y, v.get_width(), h.get_height());

               switch (v.get_content_type()) // switch sector selection
               {
                  case sector_content.text_input_display:
                     {
                     }
                     break;
                  default:
                     break;
               }
            }
         }
      }

      public override void draw_masking_sprite(Engine engine)
      {
         base.draw_masking_sprite(engine);
         // element specific functionality
         foreach (horizontal_sector h in zones)
         {
            foreach (vertical_sector v in h.get_verticals())
            {  // rendering sequence - v contains x_start and x_end of the rectangle but y_start and y_end are in the "h"
               // calculate current vertical sector rectangle in on-screen coordinates based on "h" and "v" sectors and Unit origin(x,y)
               Rectangle draw_zone = new Rectangle(v.get_xs() + (int)get_origin().X, h.get_ys() + (int)get_origin().Y, v.get_width(), h.get_height());

               switch (v.get_content_type()) // switch sector selection
               {
                  case sector_content.text_input_display:
                     {
                     }
                     break;
                     default:
                     break;
               }
            }
         }
      }
```

```
    public override void draw_post_processing(Engine engine, Color interface_color, float interface_transparency)
    {
      base.draw_post_processing(engine, interface_color, interface_transparency);
      // element specific functionality
      foreach (horizontal_sector h in zones)
      {
        foreach (vertical_sector v in h.get_verticals())
        {  // rendering sequence - v contains x_start and x_end of the rectangle but y_start and y_end are in the "h"
          // calculate current vertical sector rectangle in on-screen coordinates based on "h" and "v" sectors and Unit origin(x,y)
          Rectangle draw_zone = new Rectangle(v.get_xs() + (int)get_origin().X, h.get_ys() + (int)get_origin().Y, v.get_width(), h.get_height());

          switch (v.get_content_type()) // switch sector selection
          {
            case sector_content.text_input_display:
              {
              }
              break;
            default:
              break;
          }
        }
      }
    }
}
```

PARTICLE SYSTEM

IPARTICLECREATING INTERFACE

This interface enables any Class object implementing it to be considered an entity capable of emitting particles. It will have an Emitter class object which will be updated by the ParticleEngine each cycle. Candidates for implementation - Cloud object (for rain, snow), Tree (for leaves) in the wind, Water (for falling water), Lights (for sparkles and embers)

PARTICLE ENGINE

This is the main class for the system. It checks every registered IParticleCreating object for Emitter List, then each Emitter in that list is checked for a Particle List, finally everyParticle is updated in the Update() and rendered in Draw().

Emitter - an object of this class generates Particle objects

Particle - a partile object is the smallest piece of the system - the one that will be drawn on the screen as a base for effects and micro animations. Particles reside in the List inside ParticleEngine.

Cloud emitters are generated in the World class in the update_world

SETTING UP PARTICLE GENERATION

1. Implement IParticleCreating interface.
2. Register an object at the time of creation using : engine.get_particle_engine().register_emitter_enabled_object(temp); where temp is a newly created object. To unregister in the future - use unregister_emitter_enabled_object
3. Create emitter using engine.get_particle_engine().create_emitter and setting thehost as newly create object. Multiple coordinates are available for use in case there's more than one emitter of the same type needed.
4. Each emitter will generate particles based on: either signal from the host class or automatic generation flag. Particles will be added to a single collection inside ParticleEngine class, so they are independent of emitter and do not disappear prematurely.
5. Each particle is checked for destruction, position, maximum distance and other values. All particles are drawn in groups defined by their type, so that sprite switch overhead is minimized.

SETTING UP SYSTEM MENU FOR PARTICLE TEST CONTROL

Purpose: test and preview particle effects using various updatable properties
Process:

1. Create a container
2. Add control elements to the new container and assign newly create action type to each element
3. add container to the GUI object
4. In editor update - collect switch button value and assign it to proper handler - to actually update the value on click -in editor_command **when the action connected to the button is fired** - switch button value.
5. In editor_command - based on action - change particle type

change_test_particle_shape and change_particle_trajectory in Engine.cs control the particle shapes and movement on the click of UI button with proepr action type.

Areas where a new **trajectory type** needs to be defined (or updated):

- public **enum actions** - add an entry to enum **(skip if updating)**
- (if a button is associated) editor_command add one or more cases, then use engine.change_particle_trajectory
- (if a button doesn't exist yet) in Editor load content function create and add a button to container
- in ParticleEngine update - complete a switch case for a new trajectory

- in Engine- change_particle_trajectory needs a case added and appropriate acceleration applied

METRICS

Unit Testing

- new project > test > unit test program
- add necessary references (for this project: solution projects, find the project name then click ok)
- create test environment by :
  (arrange) creating objects and/or functions to be tested,
  (act) do any necesasry manipulations with existing objects
  (assert) using Assert class - compare expected and actual results of the function
- after all tests are green - continue the work

Performance test

analyze > performance

Particle system test

Graphics card: AMD rx380x
Particle count: 14 000
FPS: stable 60fps
Rendering time: ~10ms cycle (out of maximum 16.67 ms per frame @60 fps)
Verdict: Current system workrate ~10 000 particles at the same timevisible on-screen