TechTalk

The fundamental problems of *GUI applications* & why people choose *React*

Oliver N.
Software Engineer

TL;DR

- All applications for normal users are GUI applications.
- The fundamental problem is rendering GUI (include assembling GUI, handling user input and integrating with application logic).
- This is not another talk about React, ES2015, Webpack, and other fancy things!

Desktop applications *are* GUI applications

Desktop applications *are* GUI applications

Mobile applications *are* GUI applications

Desktop applications *are* GUI applications

Mobile applications *are* GUI applications

Web applications *are* GUI applications

Every application you use daily

is

GUI applications

Every application you use daily

is

GUI applications

(We developers* are the only ones that work with terminal**)

* Including programmers, hackers, specialists, etc.

** Even the terminal itself is a GUI application!

So if you want to build an application for normal users,

you have no choice but GUI application!

Let’s build a basic GUI application (1)

(without GUI library & framework)

Bare metal

Tier 1:

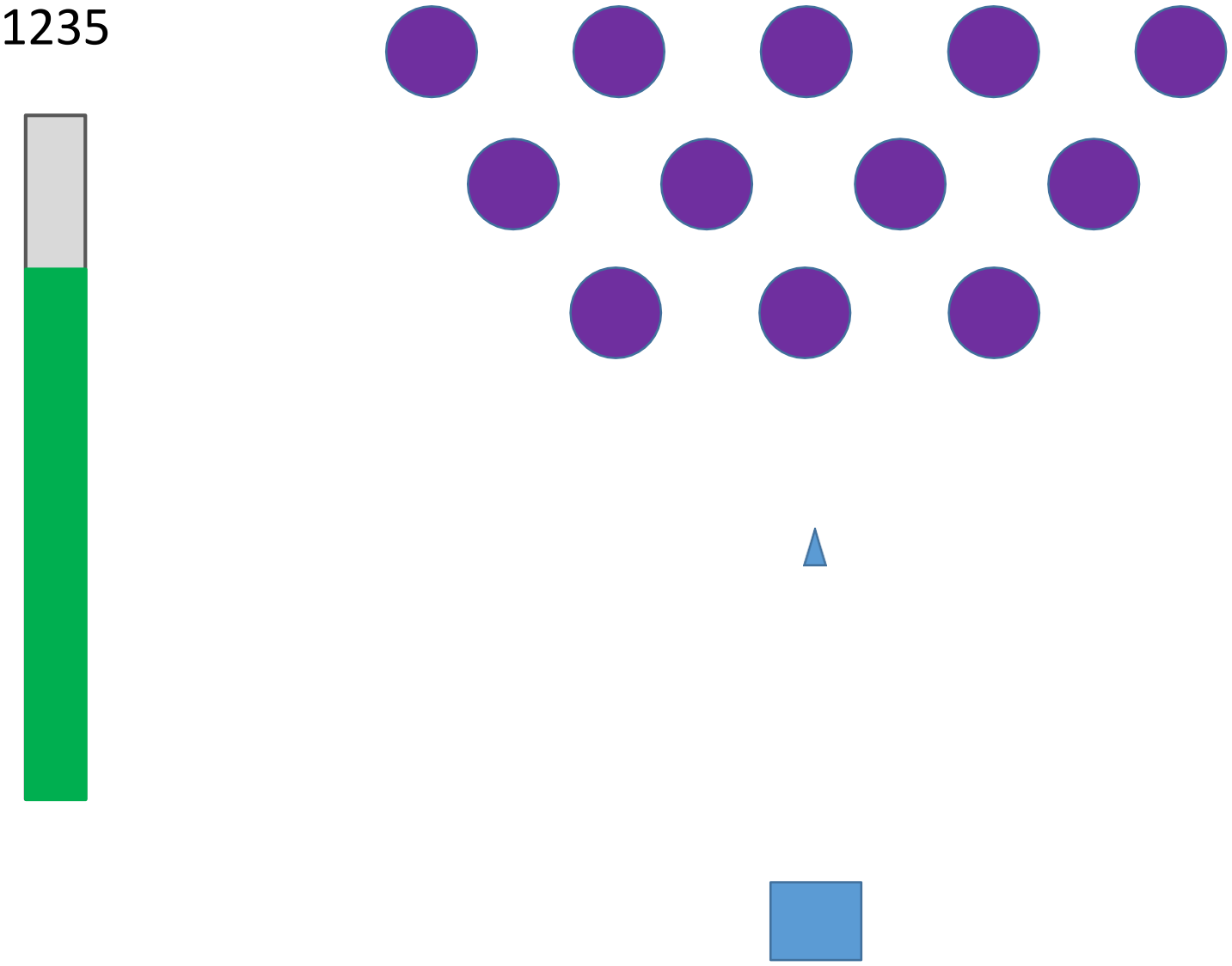
What do we have?

- A drawing buffer for output
- An input processing unit
- A programming language
- No GUI library or framework

Let’s build things from scratch*!

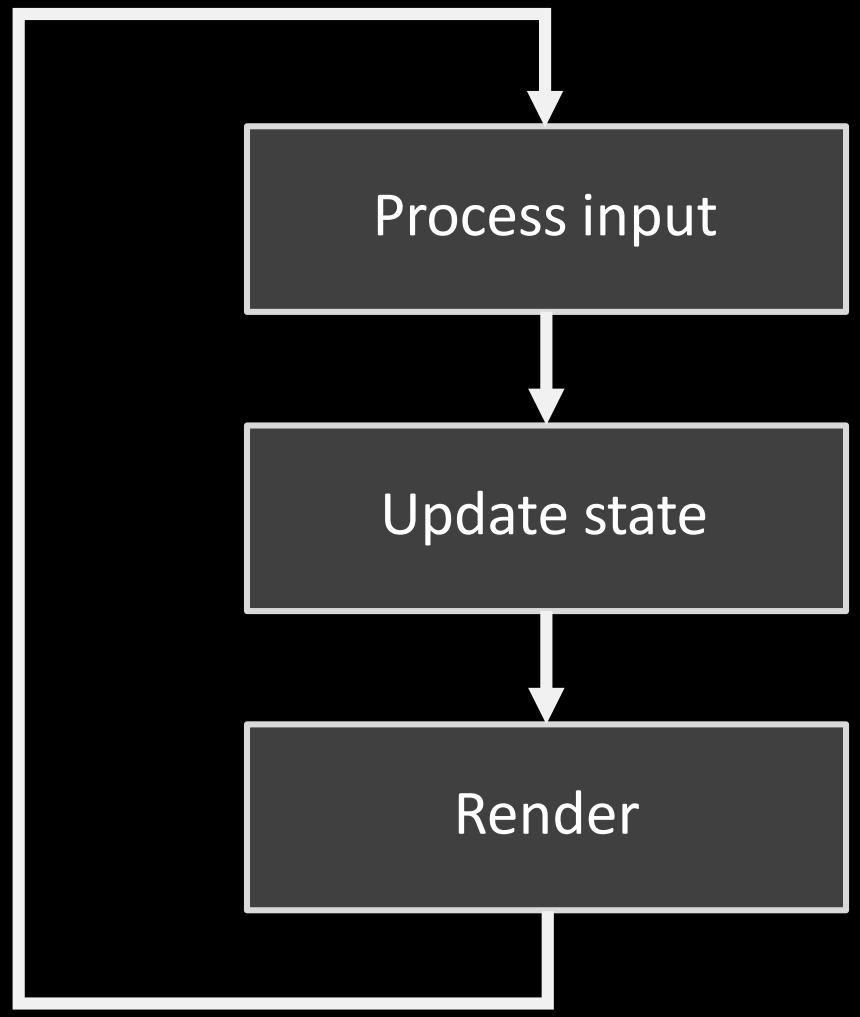
* Sample code using JavaScript.

1235



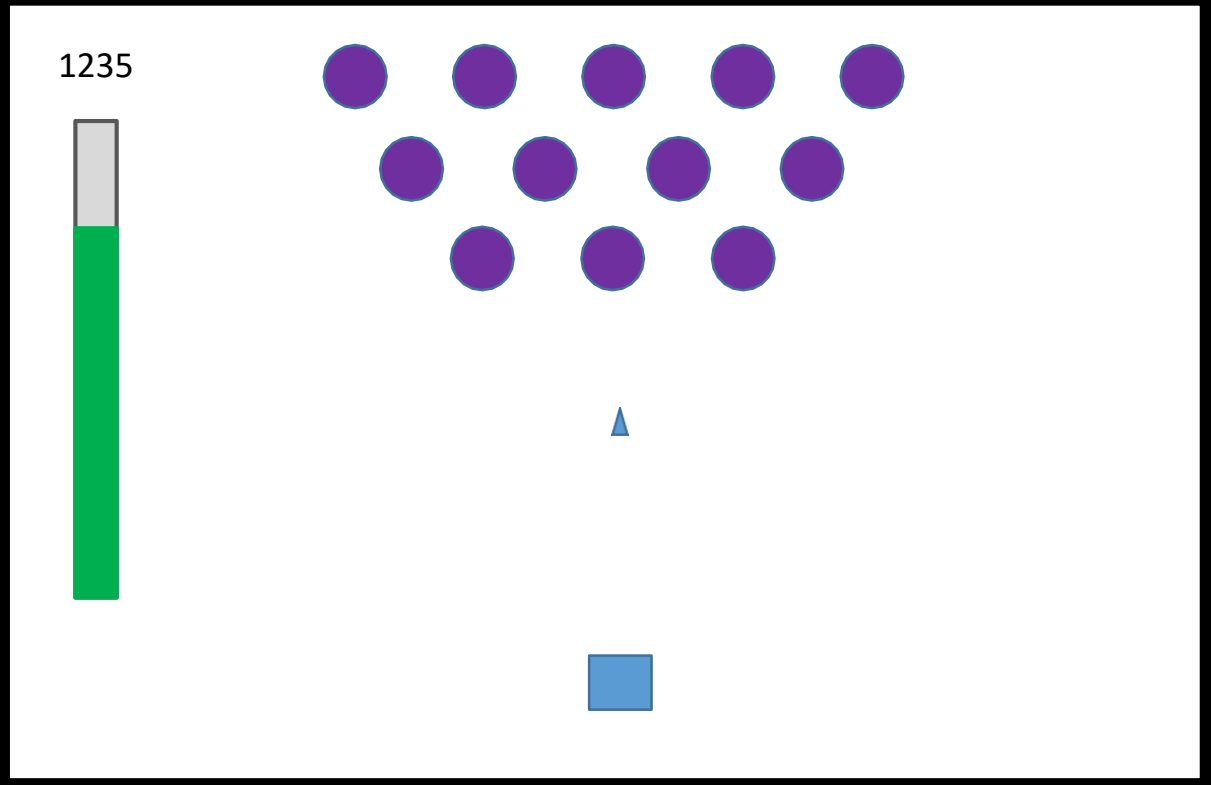
The game loop

```
while (playing) {
    processInput();
    updateState();
    render();
    waitForNextFrame();
}
```



The game state

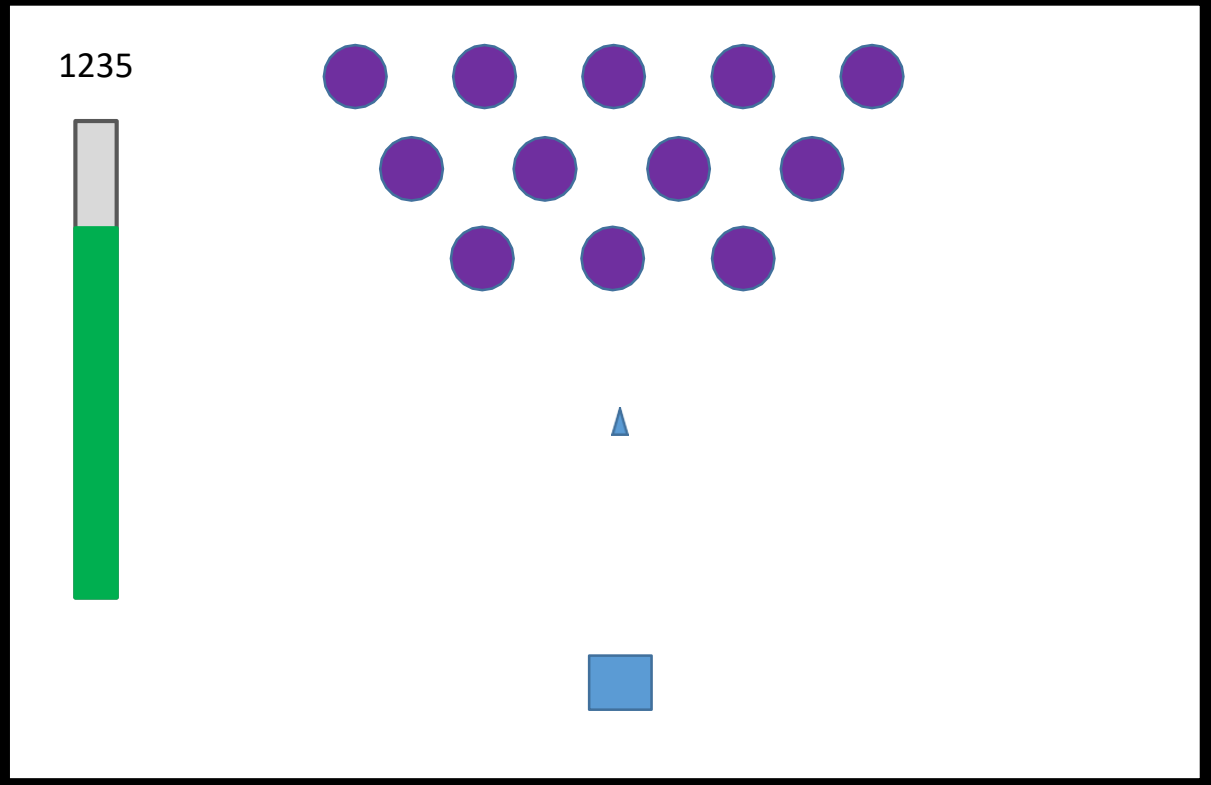
```
var gameState = {
    score: 1235,
    time: 1200,
    player: {
        x: 120,
        y: 30,
    },
    enemies: [ … ],
    bullets: [ … ]
};
```



The rendering function

```
function render() {
    renderBackground();
    renderEnemies();
    renderCharacter();
    renderBullets();
    renderScore();
    renderTimeBar();

    swapBuffer();
}
```

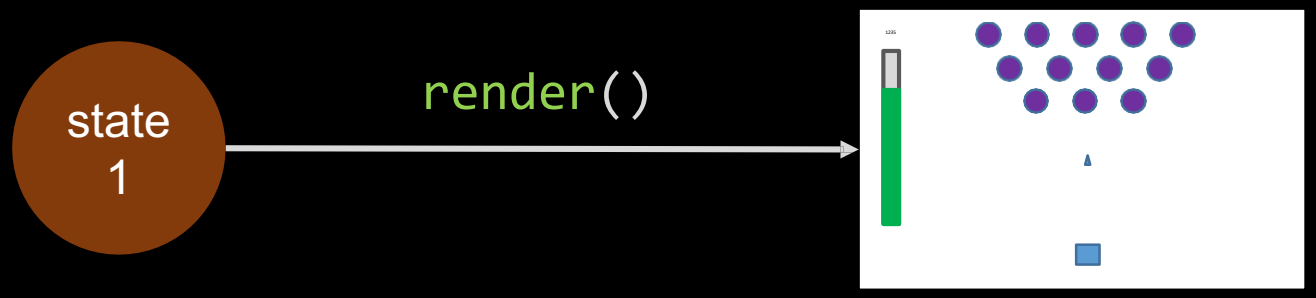


Updating state

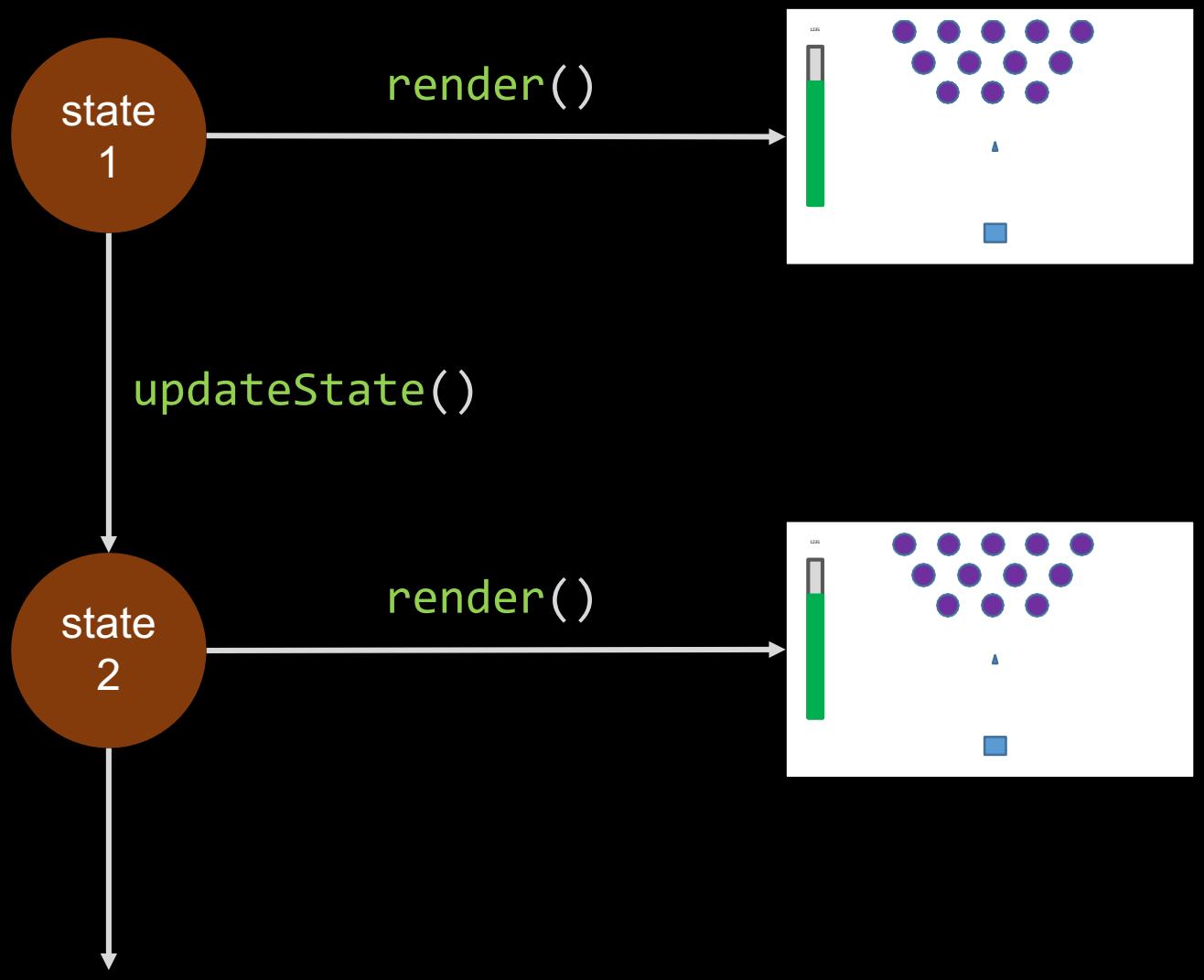
```
function updateScore(val) {
    gameState.score += val;
}

function updatePosition(dx, dy) {
    gameState.player.x += dx;
    gameState.player.y += dy;
}
```

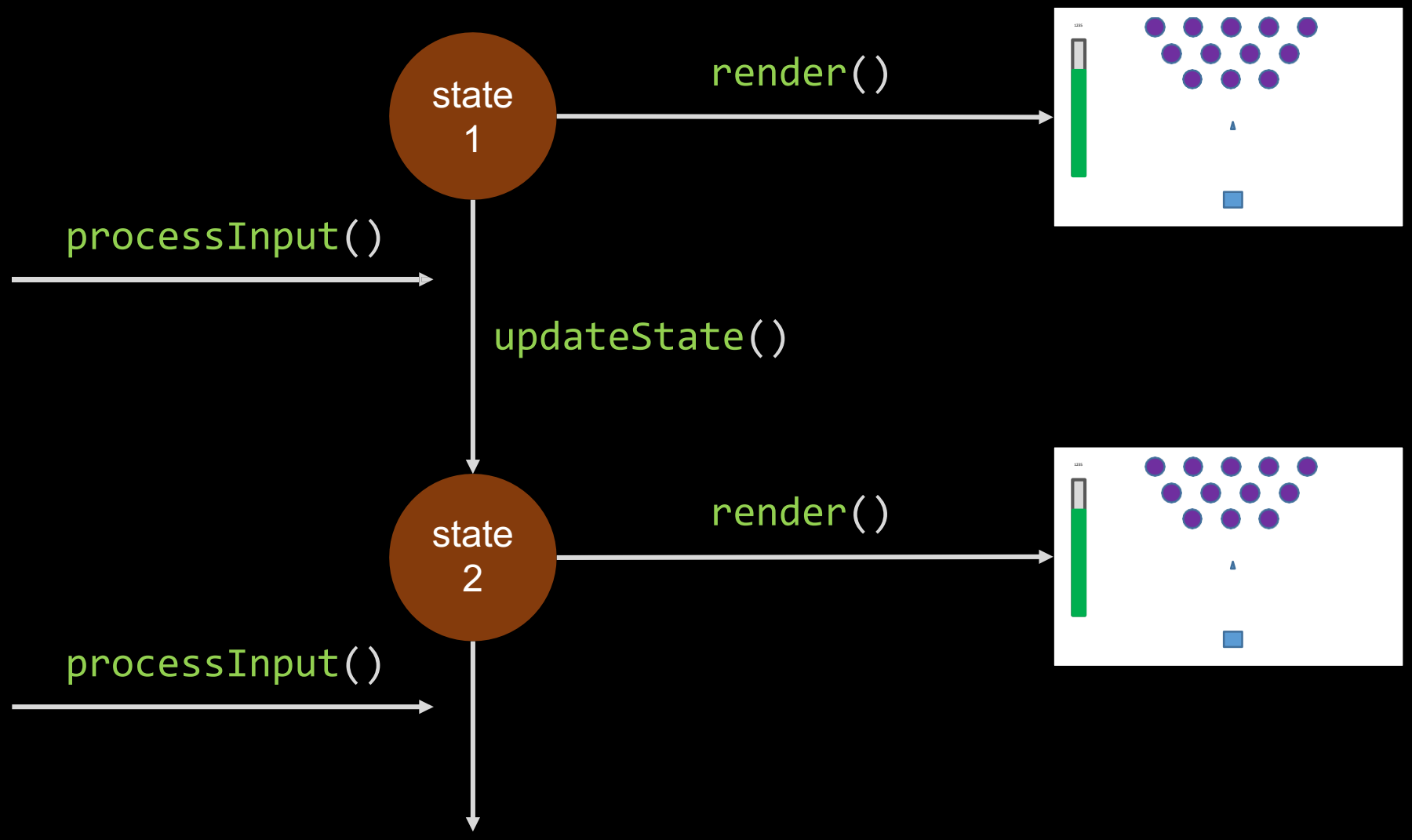
Life of a GUI application



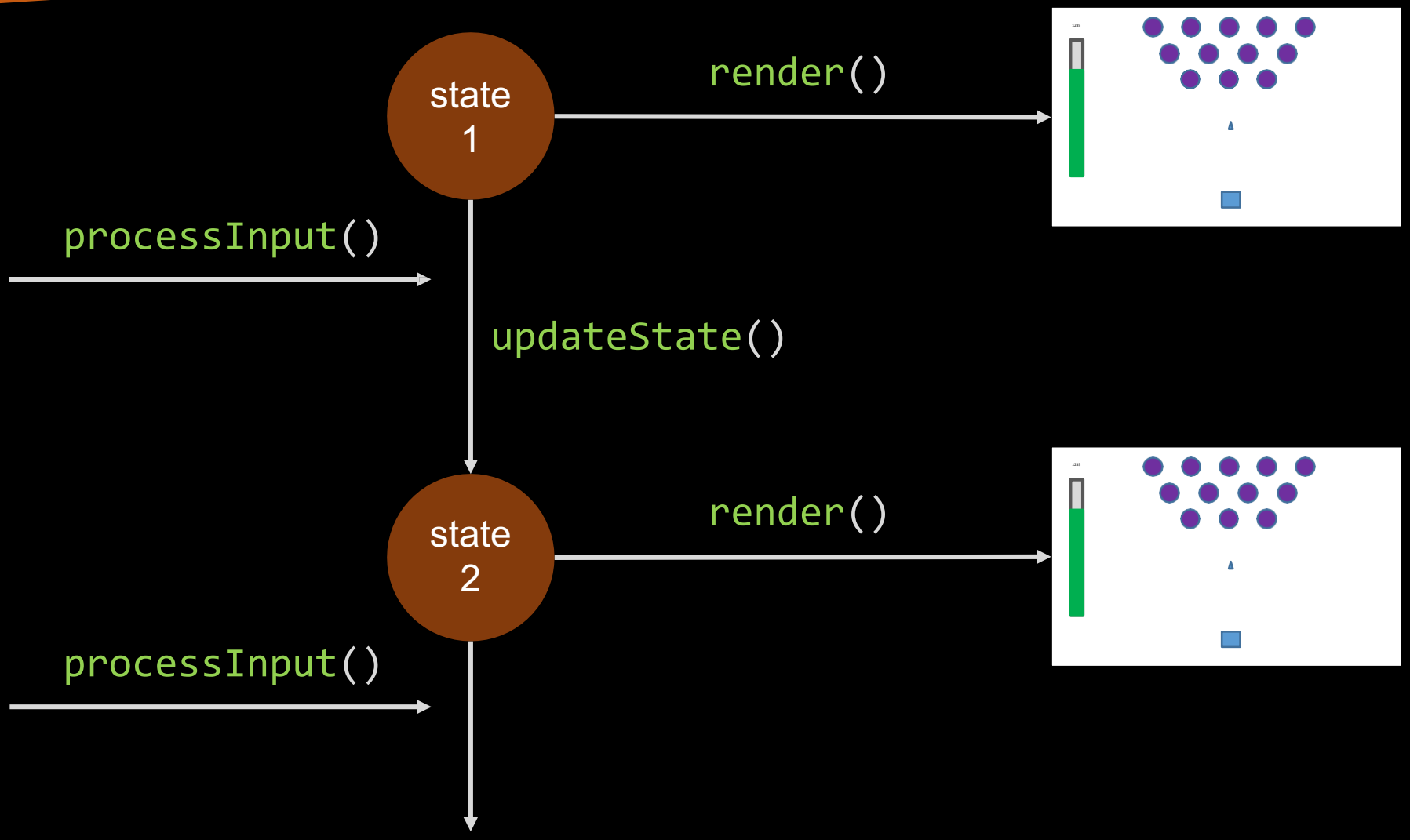
Life of a GUI application



Life of a GUI application



A state transition system



Luckily you don’t have to write “the loop” yourself.

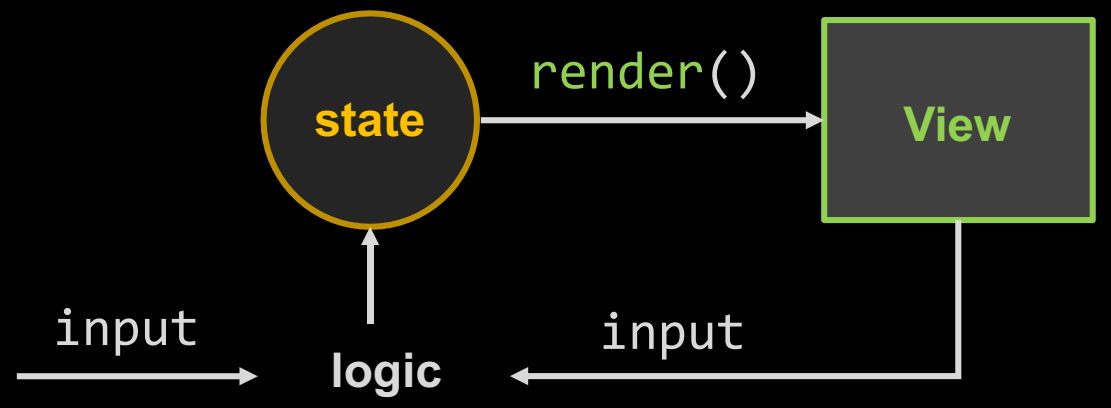
The operating system handles it for you.

The fundamental problems of GUI applications

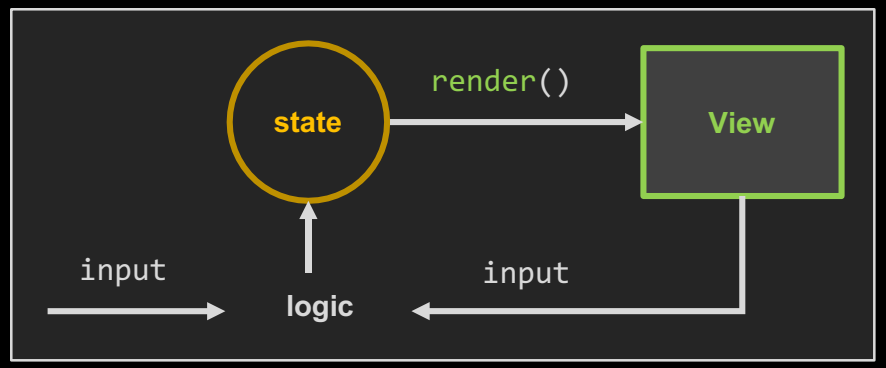
The problems

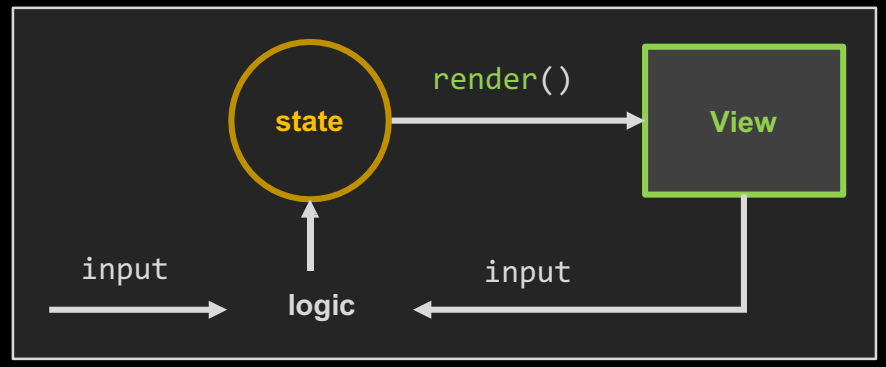
- Creating and assembling GUI
- Handling user input
- Integrating GUI & business logic

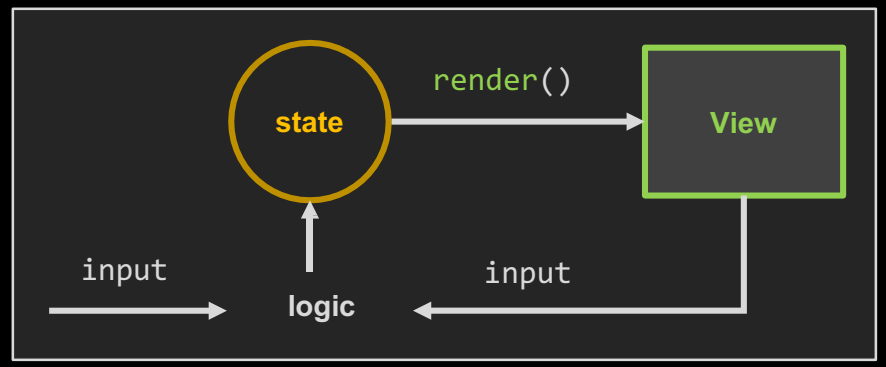
Basic GUI application architecture



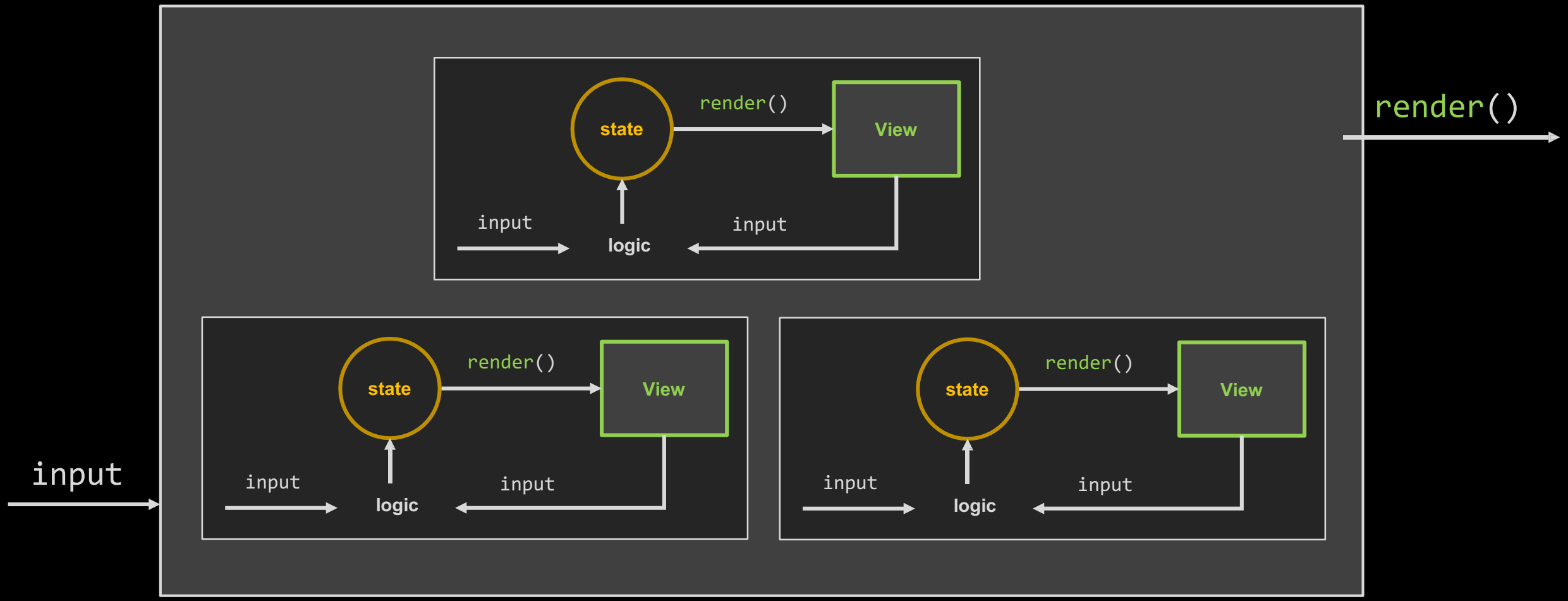
Assembling GUI







Assembling GUI



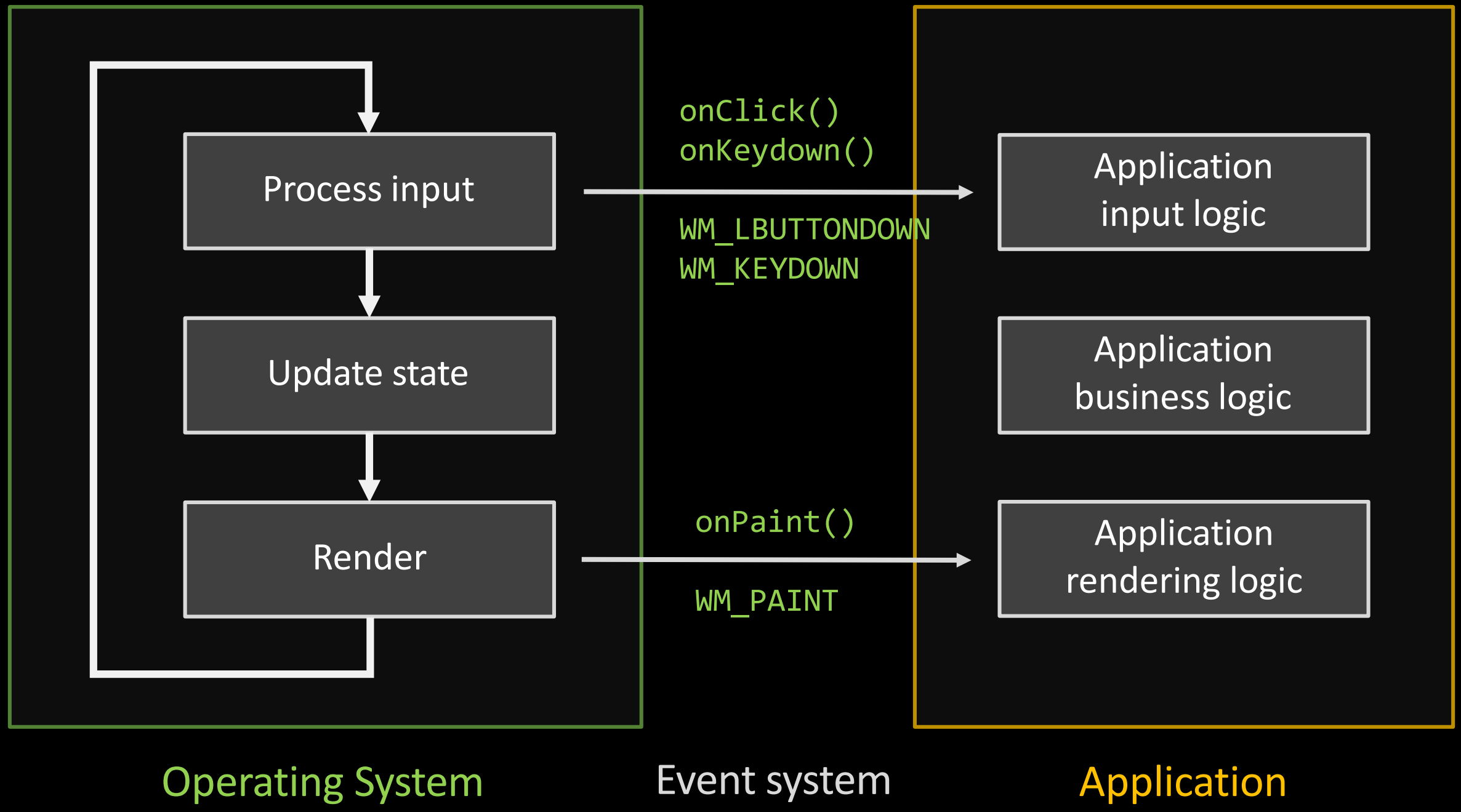
Tier 1

- Creating and assembling GUI → render() function
- Handling user input → (…)
- Integrating GUI & business logic → do it yourself

What does a modern operating system offers?

Bare metal

Tier 1: Operating System


Process input
Update state
Render
onClick()
onKeydown()
WM_LBUTTONDOWN
WM_KEYDOWN
onPaint()
WM_PAINT
Application input logic
Application business logic
Application rendering logic
Operating System
Event system
Application

What does a modern operating system offers?

- Handle “the loop”
- Process raw input and provide event system

What does a modern operating system offers?

- Creating and assembling GUI → (defer to app platform)
- Handling user input → Event system
- Integrating GUI & business logic → do it yourself

Let’s build a basic GUI application (2)

(without GUI library & framework)

Close to bare metal

Tier 2:

Tier 1: Operating System

What do we have? (at tier 2)

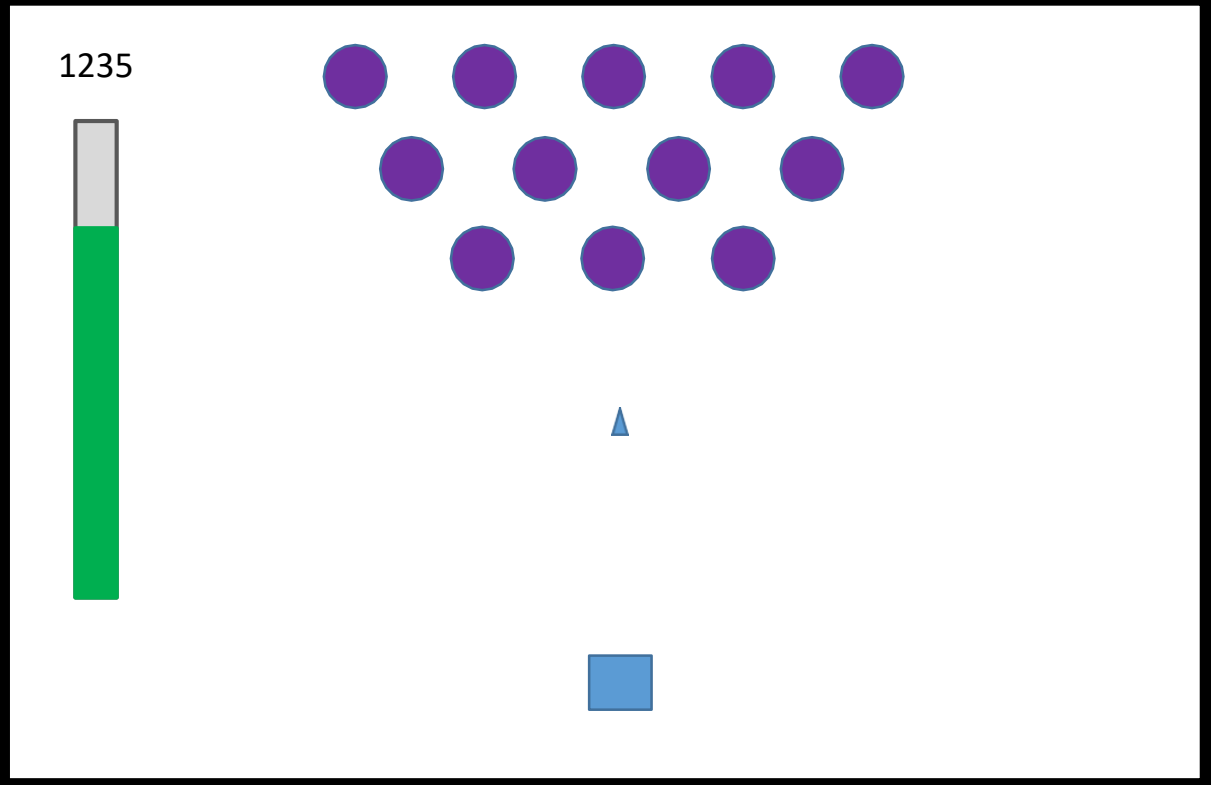
- Component system
- Event system

Sample code using Windows API*

* Win32 & COM API. Read more:
https://msdn.microsoft.com/en-us/library/windows/desktop/ff381399(v=vs.85).aspx

The application state

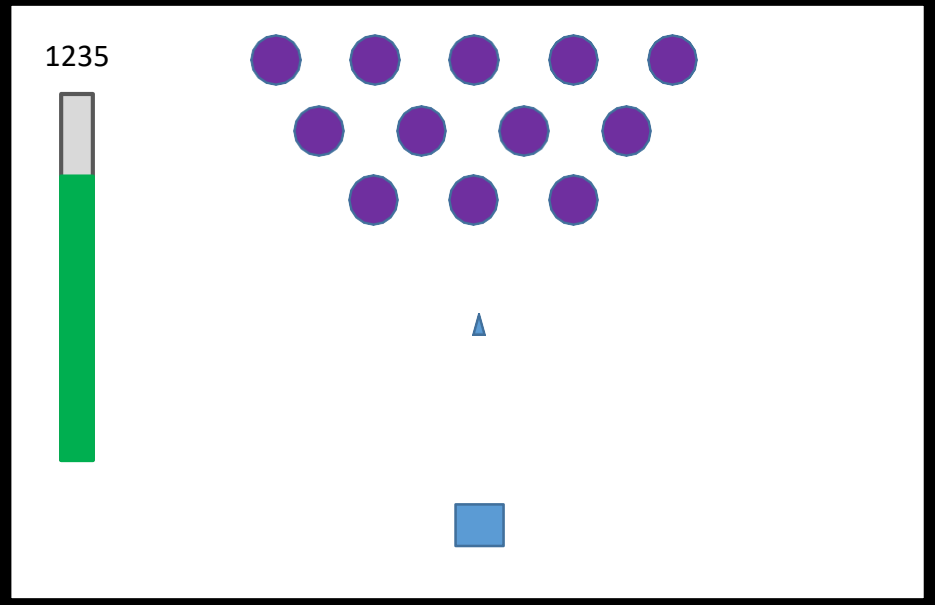
```
struct {
      int score,
      int time,
      PLAYER player,
      ENEMIES enemies,
      BULLETS bullets
} gameState;
```



The rendering function

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT
  message, WPARAM wParam, LPARAM lParam)
  PAINT_STRUCT ps;
  HDC hdc;
  switch (message) {
    case WM_PAINT:
      hdc = BeginPaint(hWnd, &ps);
      // ...

      EndPaint(hWnd, &ps);
      break;
  }
};
```



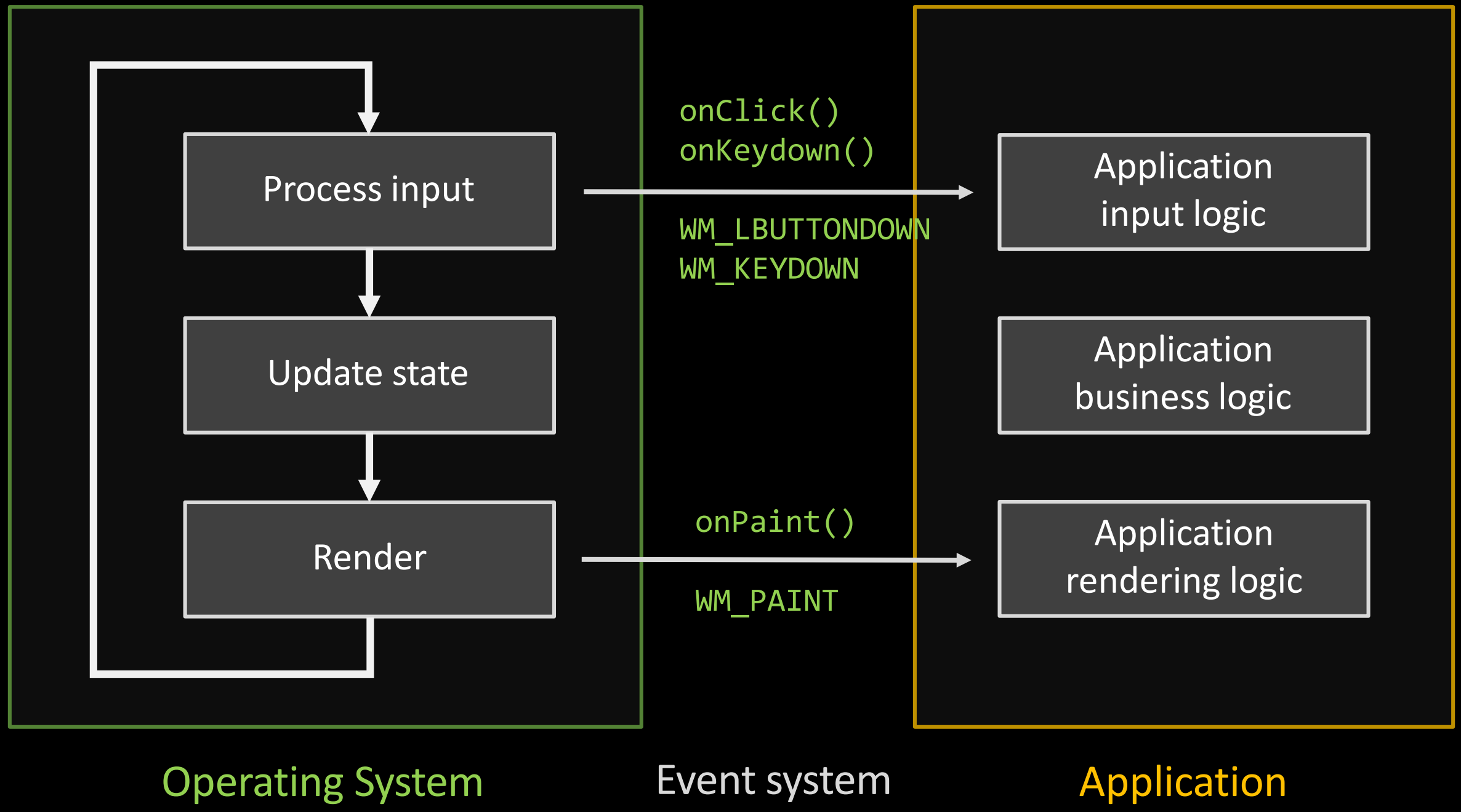
Composing components

- Create child windows
- Attach them to the app window
- In response to WM_PAINT:
  - Pass WM_PAINT to child windows

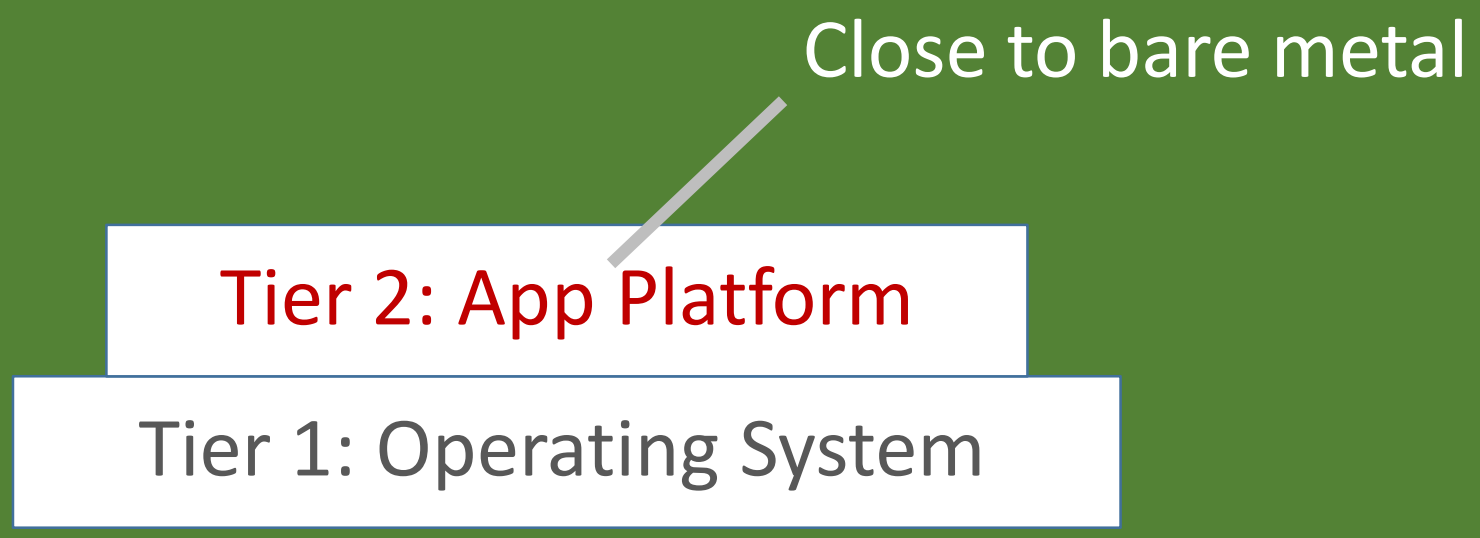
Handling input

- Handling user input

  Response to input events `WM_LBUTTONDOWN`, `WM_KEYDOWN`
- Handling application life cycle `WM_CREATE`, `WM_DESTROY`


Process input
Update state
Render
onClick()
onKeydown()
WM_LBUTTONDOWN
WM_KEYDOWN
onPaint()
WM_PAINT
Application input logic
Application business logic
Application rendering logic
Operating System
Event system
Application

What does a modern application platform offers?



Android Platform

- Composing elements: XML Layout, GUI components
- Handling user input: Event system
- Integrating business logic: Callback

```
<Button
    xmlns:android="http://schemas..."
    android:id="@+id/button_send"
    android:text="@string/button_send"
    android:onClick="sendMessage" />
```

```
public void sendMessage(View view) {
  // Do something
}
```

http://developer.android.com/guide/topics/ui/controls/button.html

Windows Presentation Foundation (WPF)

- Composing elements: XAML, GUI components
- Handling user input: Event system
- Integrating business logic: Handler

```
<Button
  Grid.Column="1" Grid.Row="3"
  Margin="0,10,0,0" Width="125"
  Height="25" HorizontalAlignment="Right"
  Click="Button_Click">View</Button>
```

```
private void Button_Click(
  object sender, RoutedEventArgs e) {
  // Do something
}
```

https://msdn.microsoft.com/en-us/library/mt270964.aspx

Web Platform (HTML & JS)

- Composing elements: HTML, GUI components
- Handling user input: Event system
- Integrating business logic: Callback

```
<button
  style="width:100px;height:40px"
  onclick="sayHello()">
  Say Hello
</button>
```

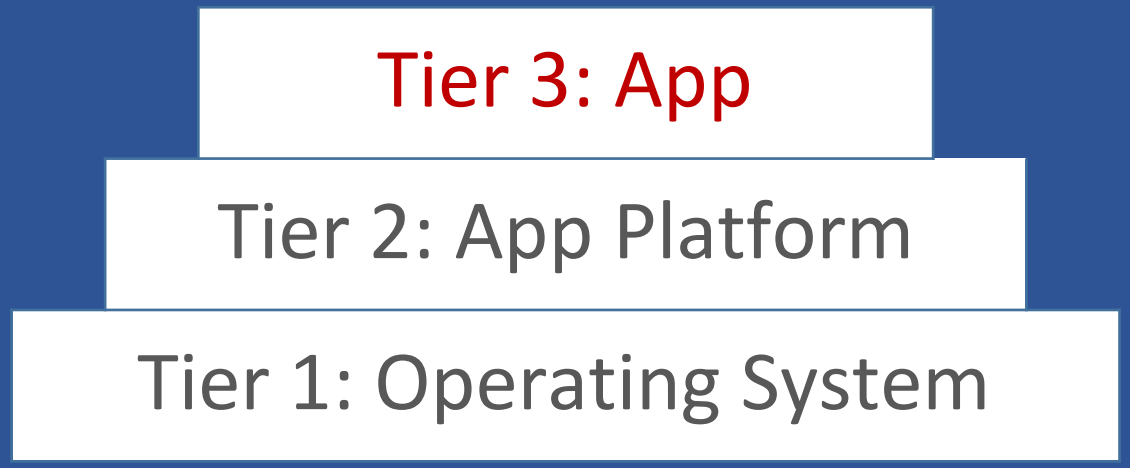
```
function sayHello(e) {
  // Do something
}
```

What does an application platform offers?

- Creating and assembling GUI → Pre-built components, Domain specific language (DSL) for GUI
- Handling user input → Event system
- Integrating GUI & business logic → Callback, set state

Let’s build a basic GUI application (3)

(without GUI library & framework)



What do we have? (at tier 3, HTML & JS)

- DSL & pre-built GUI components
- Event system
- Callback & set component state
- No GUI library or framework.

We still want to create our custom components!

Let’s build a TODO application.

The application state

```
var appState = {
  todos: [{
    title: “hello”,
    complete: false
  }, {
    title: “world”,
    complete: false
  }]
};
```

Updating state

```
function addTodo(label) {
  appState.todos.push({
    title: label,
    completed: false
  });
}

function toggle(index) {
  var item = appState.todos[index];
  item.completed = !item.completed;
}
```

The rendering function

```
function render() {
  // !?
}
```

The rendering function – First try

```
function render() {
  var $listTodos = document.getElementById("todos")
  for (var i=0; i < appState.todos; i++) {
    // update, insert or delete DOM elements
  }

  var $numActive = document.getElementById("num-
  active")
  $numActive.innerHTML = getNumActive();

  // ...
}
```

So far, we have defined application state and logic just fine.

The only hard part that kept us back is rendering step.

(Include generating HTML, keeping updated with app state and registering event callbacks)

The rendering function – Second try

```
var lastState;  // store last appState for comparing
function render() {
  var $listTodos =
  document.getElementById("todos") for (var i=0; i
  < appState.todos; i++) {
    // update, insert or delete DOM elements
  }

  var $numActive = document.getElementById("num-
  active")
  $numActive.innerHTML = getNumActive();

  // ...
  lastState = deepClone(appState);
```

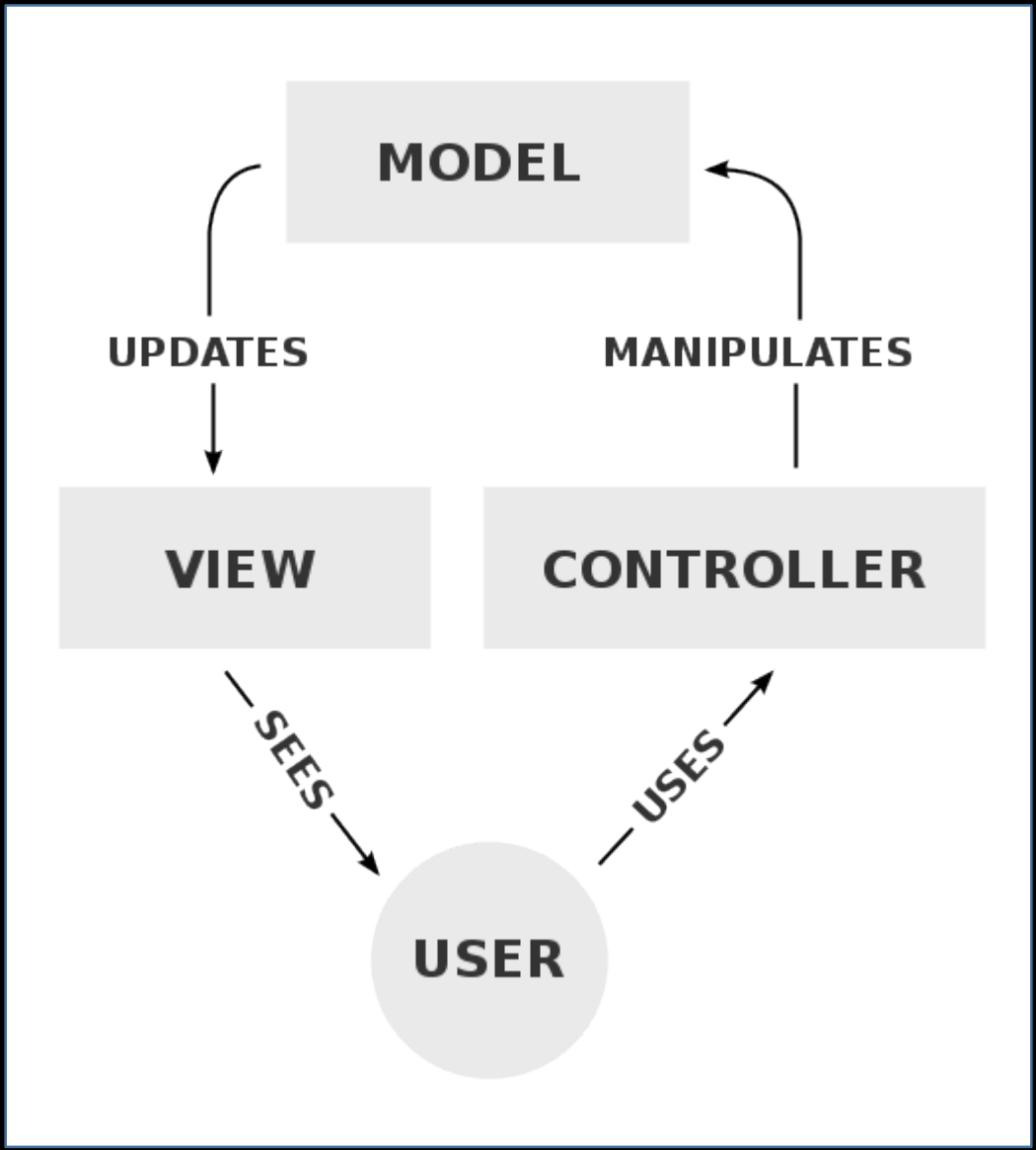
We have tried storing last application state for rendering only changed parts.

This is what frameworks like Angular.js or Backbone (Underscore) offers.

Enter
MVC & MVP

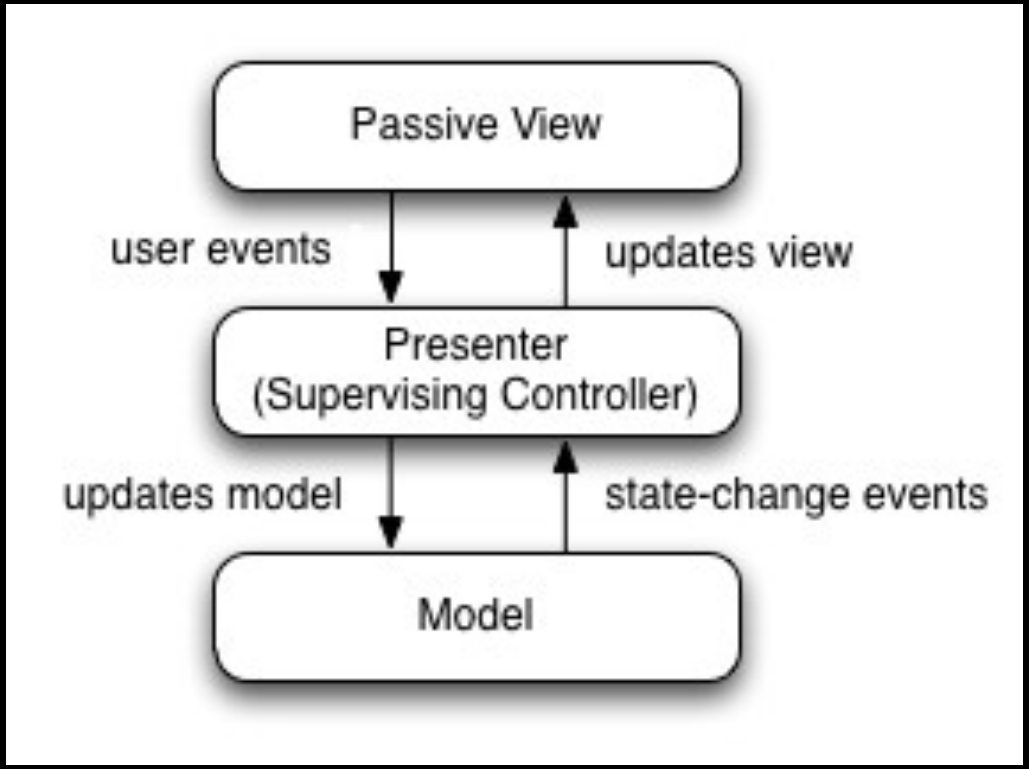
MVC

- Applying Separation of Concern to GUI applications.
- Input event
  → Controller
  → Model
  → View



MVP

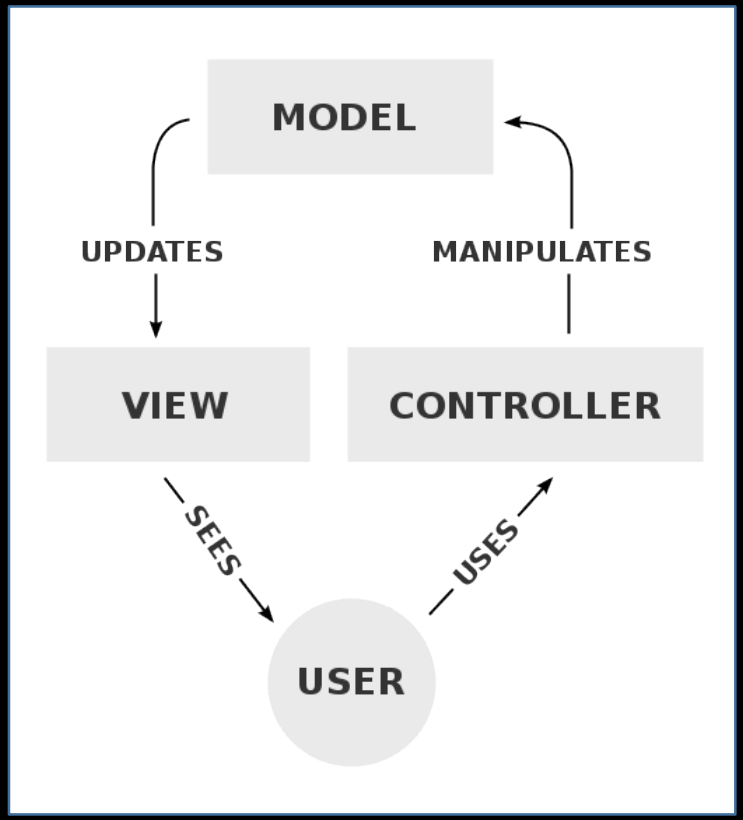
- Applying Separation of Concern to GUI applications.
- Model → Presenter → View
- View → Presenter → Model



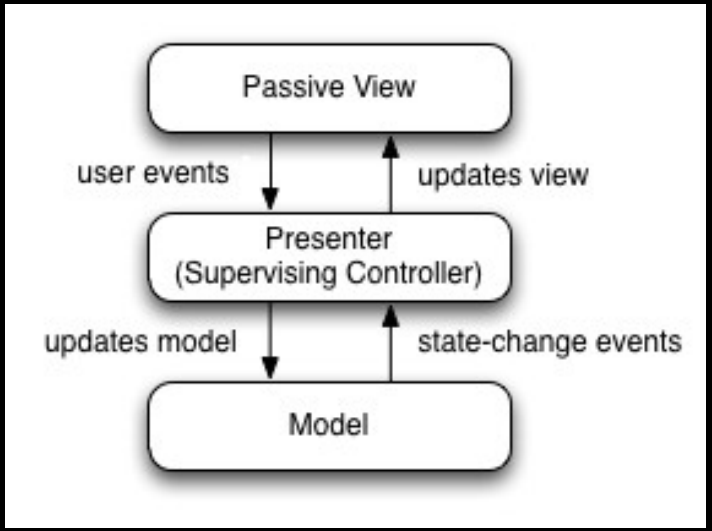
Backbone.js

```
var Todo = Backbone.Model.extend({
  default: { title: "", complete: false },
  toggle: function() {
    this.save(…);   // trigger "change" (model)
  }
});

var TodoView = Backbone.View.extend({
  template: …,
  events: …,        // callback to manipulate model (handled by controller)
  initialize: {
    this.listenTo(this.model, "change", this.render); // listen to "change"
  },
  render: function() { … },  // rendering function (view)
  // …
});
```



Angular.js (version 1)



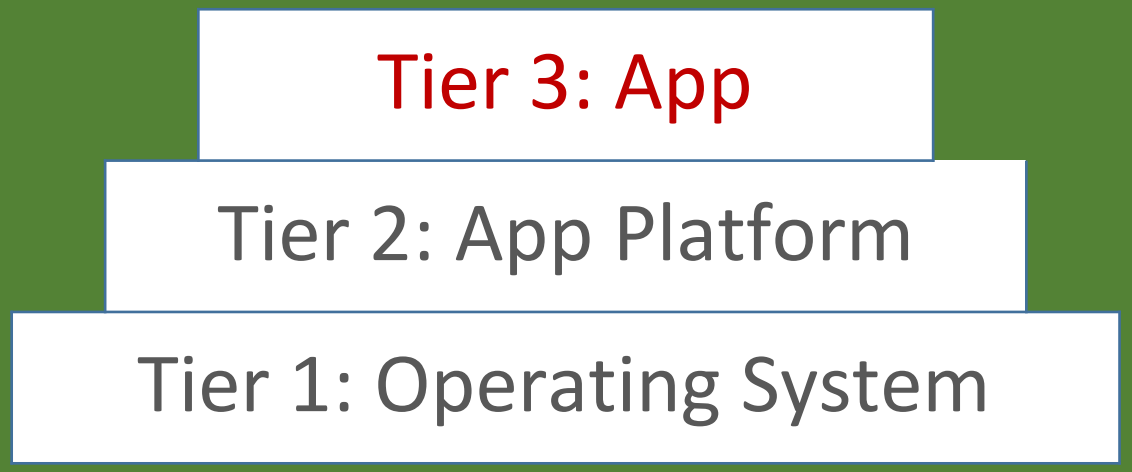

```
function TodoCtrl($scope) {
  var todos = $scope.todos = [];

  $scope.addTodo = function() {  // user event
    todos.push({ title: $scope.newTodo, completed: false }); // update state
  };

  $scope.$watchCollection("todos", function() {  // state-change event
    // …
  });
}

<form id="todo-form" ng-submit="addTodo()"> … </form>
<ul id="todo-list">
  <li ng-repeat="todo in todos"> … </li>
</ul>
```

What does a application MV* framework offers?



What does a MV* framework offers?

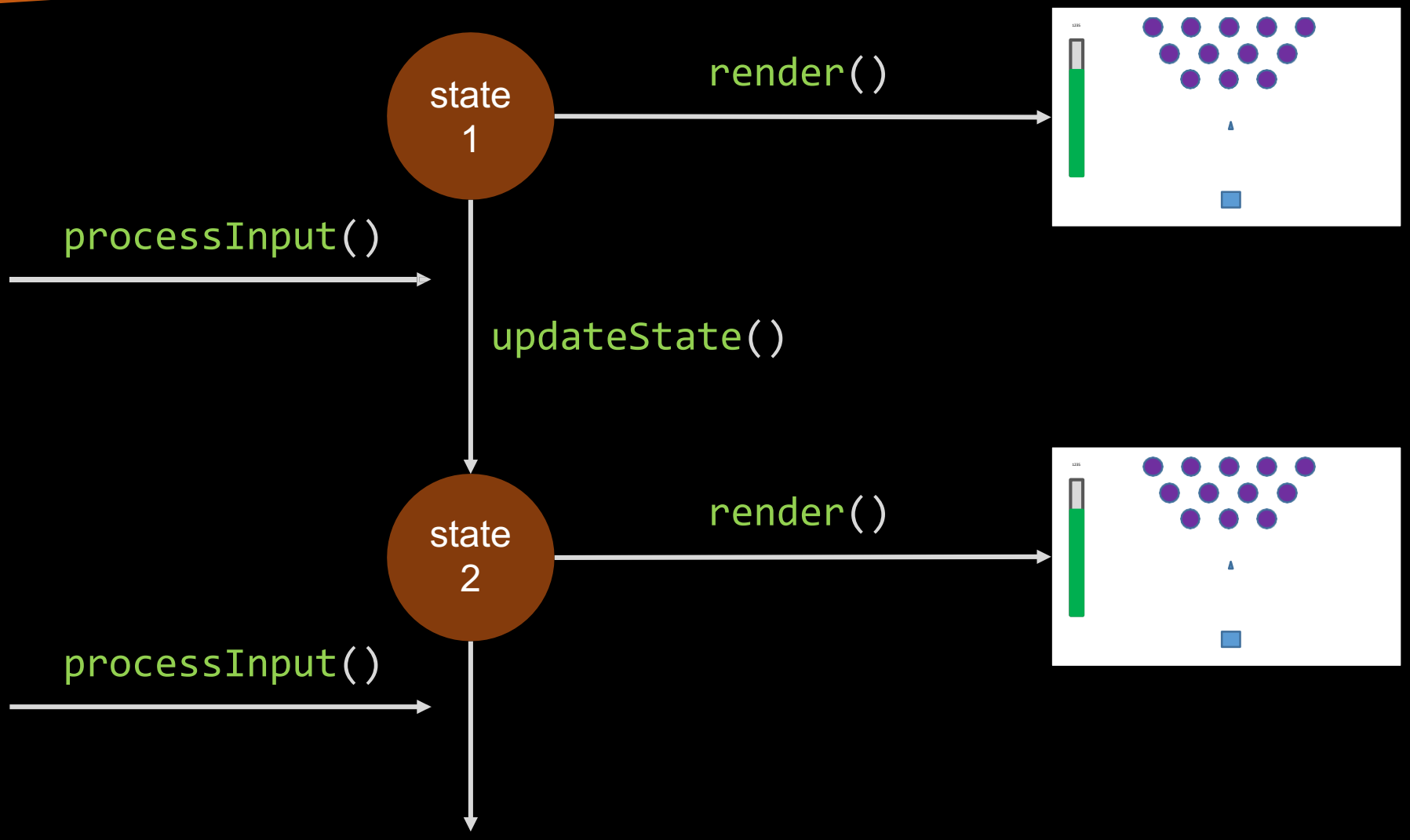
- Creating and assembling GUI → view, template
- Handling user input → user event
- Integrating GUI & business logic
  → state-change event (Backbone.js, Angular.js)

Wow, so many concepts!

model, view, template, controller, presenter,
user event, state-change event

Let’s return to our starting architecture

A state transition system



Enter React solution

Let’s put aside the fancy ways to define application state.

The only hard part is rendering step.

The application state

```
var appState = {
  todos: [{
    title: “hello”,
    complete: false
  }, {
    title: “world”,
    complete: false
  }]
};
```

Updating state

```
function addTodo(label) {
  appState.todos.push({
    title: label,
    completed: false
  });
}

function toggle(index) {
  var item = appState.todos[index];
  item.completed = !item.completed;
}
```

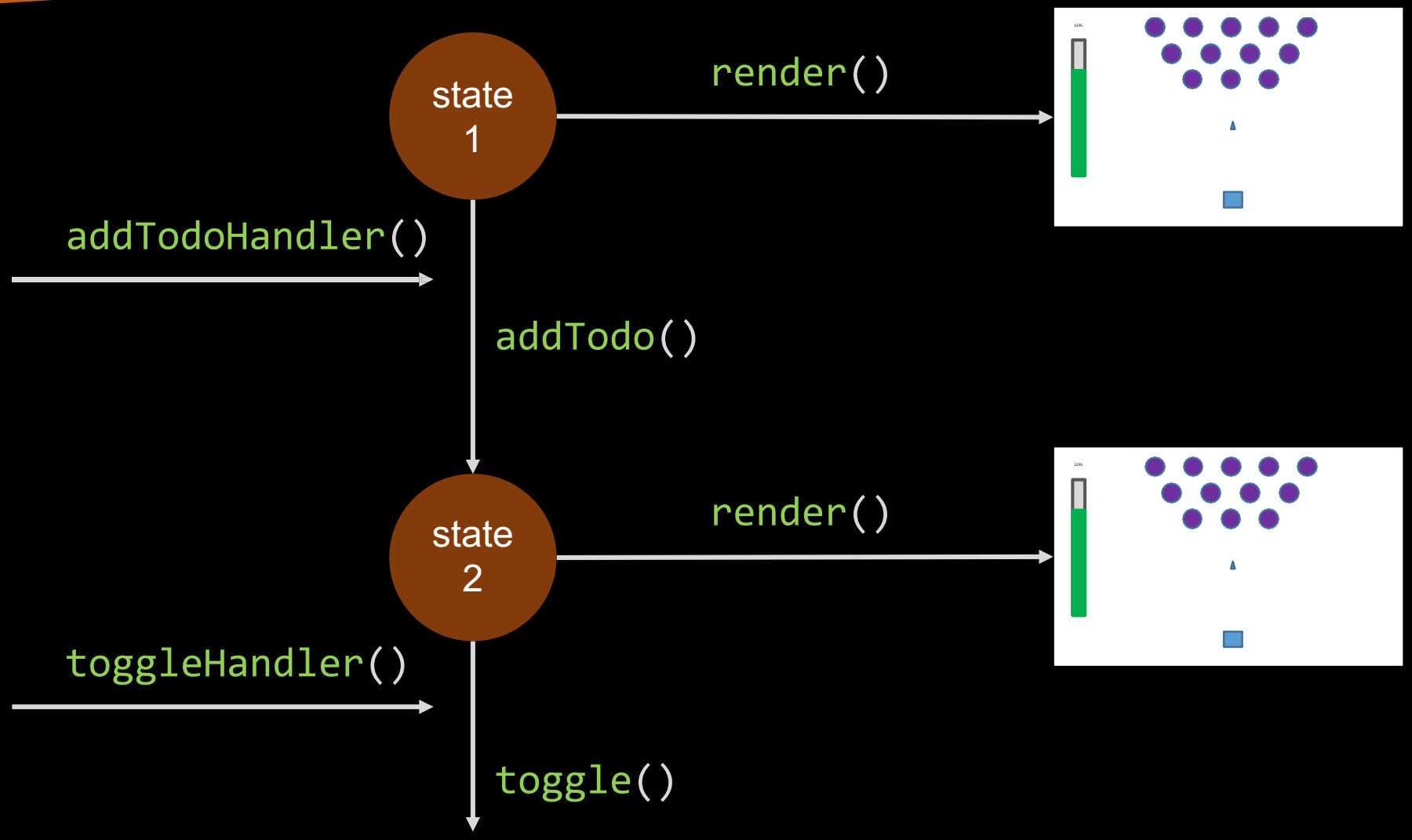
The rendering function

```
React.createClass({
  render: function() {
    return (
      <ul>
      {
        appState.todos.map(function(item) {
          return <li> { item.title } </li>;
        })
      }
      </ul>
    );
  }
});
```

Handle state change

```
React.createClass({
  addTodoHandler: function() {
    var label = this.refs.inputTodo.value;
    addTodo(label);       // update application state
    this.forceUpdate();   // trigger rendering function
  },
  render: function() {
    return (
      <div>
        <input ref="inputTodo"/>
        <button onClick={this.addTodoHandler}/>
        <ul> … </ul>
      </div>
    );}
});
```

A state transition system



React lets us work with our classic architecture and helps solving the hard part: rendering!

No need to rewrite our application in an opinion way.

We only need to understand 2 functions to start working with React:

- forceUpdate()
- render()

What does React offers?

- Creating and assembling GUI → React components
- Handling user input → user events
- Integrating GUI & business logic
  → keep GUI updated when application state changed

Why do people choose React?

What do people choose React?

- As we see, the only hard part is rendering step.
- React is a view library. It solves the right problem and solves it well.
- It leaves application state for us. This is good because:
  - We work with classic architecture of GUI applications.
  - We can choose which architecture works best for us.
  - We can migrate legacy applications to React without changing so much code.

How to choose a library or framework?

1. Write the prototype by your own without using library or framework.
2. Understand what them offer.
3. Choose only which ones you need.
4. Keep in mind the design that you can switch to another library later.

What’s next?

- Redux, an application state solution for React.
  - Because we understand how to handle application state, we can decide to use Redux or not. It’s up to you.

TechTalk

THANK YOU

Oliver N.
Software Engineer

TechTalk

The fundamental problems of *GUI applications* & why people choose *React*

Oliver N.
Software Engineer