Open Source Snake 360

Evelyn Wijaya–2015730030

1 Data Skripsi

Pembimbing utama/tunggal: **Pascal Alfadian Nugroho**
Pembimbing pendamping: -
Kode Topik : **PAN4502**
Topik ini sudah dikerjakan selama : **1** semester
Pengambilan pertama kali topik ini pada : Semester **45 - Ganjil 18/19**
Pengambilan pertama kali topik ini di kuliah : **Skripsi 1**
Tipe Laporan : **B -** Dokumen untuk reviewer pada presentasi dan **review Skripsi 1**

2 Latar Belakang

Snake merupakan sebuah permainan yang pertama kali dibuat oleh Peter Trefonas pada tahun 1978. Konsep *Snake* berasal dari permainan arkade yaitu *Blockade*. Awalnya *Snake* hanya dapat dimainkan pada komputer pribadi. Namun pada tahun 1997, *Snake* dapat dimainkan pada telepon genggam *Nokia*[1]. Cara bermain *Snake* adalah pemain menggerakan ular pada sebuah labirin. Ular tersebut harus mendapatkan makanan sebanyak-banyaknya tanpa menabrak dinding atau ular itu sendiri. Setiap memakan makanan, tubuh ular akan memanjang dan pemain akan semakin sulit untuk menggerakan ular tersebut dengan bebas karena tubuh ular semakin lama akan menutupi labirin tersebut.

HTML(*Hyper Text Markup Language*) adalah sebuah bahasa markah yang digunakan untuk membuat halaman web. HTML5 merupakan HTML versi 5 yang terbaru dan penerus dari HTML4, XHTML1, dan DOM level 2 HTML. HTML5 memiliki beberapa elemen baru, salah satunya adalah HTML5 Canvas. HTML5 Canvas adalah tempat untuk menggambar *pixel-pixel* yang dapat ditulis menggunakan bahasa pemrograman *JavaScript*. *Javascript* adalah bahasa pemrograman tingkat tinggi yang digunakan untuk membuat halaman web menjadi lebih interaktif. *GitHub* adalah layanan *web hosting* bersama untuk proyek pengembangan perangkat lunak yang menggunakan sistem *version control* yaitu *Git*. Dengan adanya *Github*, *programmer* dapat mengetahui perubahan yang pada *repository* tersebut.

Pada permainan *Snake*, umumnya pergerakan ular hanya atas, bawah, kiri, dan kanan saja. Pada skripsi ini, penulis akan membuat permainan *Snake* yang ularnya dapat bergerak ke segala arah dan orang lain dapat menambahkan labirin menggunakan mekanisme *pull request Github*. Dengan begitu, orang lain dapat menambahkan labirin sesuai dengan keinginanya dan pemain tidak akan cepat bosan karena labirin yang disediakan cukup banyak dan variatif.

3 Rumusan Masalah

Rumusan dari masalah yang akan dibahas pada skripsi ini adalah sebagai berikut:

- Bagaimana membangun permainan *Snake* menggunakan HTML5?
- Bagaimana cara menyimpan labirin pada file eksternal?
- Bagaimana cara menggunakan *pull request* pada *Github* agar orang lain dapat menambahkan labirin?

[1] https://en.wikipedia.org/wiki/Snake_(video_game_genre)

4 Tujuan

Tujuan-tujuan yang hendak dicapai melalui penulisan skripsi ini adalah sebagai berikut:

- Dapat membangun permainan *Snake* menggunakan HTML5.
- Dapat menyimpan labirin pada file eksternal.
- Dapat menggunakan *pull request* pada *Github* agar orang lain dapat menambahkan labirin.

5 Detail Perkembangan Pengerjaan Skripsi

Detail bagian pekerjaan skripsi sesuai dengan rencana kerja/laporan perkembangan terakhir :

1. **Melakukan studi literatur tentang HTML5, *JavaScript*, *jQuery* dan *Git*.**
   Status : Ada sejak rencana kerja skripsi.
   Hasil :

   - **HTML5 *Canvas***

     HTML5 Canvas adalah sebuah daerah *bitmap* yang dapat dimanipulasi oleh *Javascript*. Pada daerah *bitmap* tersebut, *pixel-pixel* akan di*render* oleh canvas. Setiap *frame*, HTML5 Canvas akan menggambar pada area *bitmap* tersebut menggunakan *Canvas* API(*Application Programming Interface*) yang dipanggil pada *Javascript*. API dari HTML5 Canvas yang umum adalah 2D *Context*. Dengan adanya 2D *Context*, *programmer* dapat membuat bentuk 2D, menampilkan gambar, *render* tulisan, memberi warna, membuat garis dan kurva, dan manipulasi *pixel*. HTML5 Canvas tidak hanya digunakan untuk menggambar dan menampilkan gambar serta tulisan. HTML5 Canvas dapat digunakan untuk membuat animasi, aplikasi pada *web* dan permainan.

     Untuk menambahkan *canvas* pada halaman HTML, diperlukan *tag* <canvas>. Di bawah ini adalah potongan kode untuk menambahkan *canvas* pada halaman HTML.

```
1    <canvas id='canvas' width='500' height='300'>
2        Your browser does not support HTML5 Canvas.
3    </canvas>
```

Listing 1: Menambahkan *canvas*

     Diantara tag <canvas> dan </canvas>, dapat dituliskan text yang akan ditampilkan jika browser tidak support HTML5 Canvas.

   - ***Javascript***

     Javascript adalah bahasa pemrograman yang ringan, *interpreted* dan berorientasi objek yang digunakan pada halaman *web*. *Javascript* dapat membuat objek dengan menambahkan *method* dan atributnya sama seperti bahasa pemrograman C++ dan *Java*. Setelah objek diinisialisasi, maka objek tersebut dapat dijadikan *blueprint* untuk membuat objek lain yang mirip. *Javascript* dapat digunakan untuk mengimplementasi hal yang kompleks pada halaman web. Contohnya adalah menamplikan peta yang interaktif dan membuat animasi 2D/3D. Selain *Javascript*, HTML(*HyperText Markup Language*) dan CSS(*Cascading Style Sheet*) merupakan bagian/komponen penting dalam pembuatan halaman *web*.

Untuk menambahkan Javascript pada sebuah halaman web yang dibuat, gunakan tag <script>. Ada 2 cara untuk menambahkan Javascript yaitu menambahkan langsung di halaman web tersebut(Internal Javascript) dan menambahkan file Javascript terpisah(External Javascript).

Variabel

Variabel adalah sebuah wadah untuk menyimpan nilai/*value*. Untuk mendeklarasi variable pada *Javascript*, digunakan *keyword 'var'*. Variabel pada Javascript tidak perlu menuliskan tipe datanya ketika mendeklarasikan variabel. Hal ini dikarenakan variabel pada Javascript dapat mencakup semua tipe data. Di bawah ini adalah potongan kode untuk mendeklarasikan variabel.

```
var myVariable = 3;
```

Listing 2: Deklarasi variabel

Constant

Constant adalah sebuah variabel *read-only*, artinya nilai pada *constant* tidak dapat diubah. Untuk mendeklarasikan *constant*, digunakan *keyword 'const'*.

Function

Function adalah sekumpulan perintah/*statements* untuk menjalankan suatu tugas atau menghitung nilai. Untuk membuat *function*, digunakan *keyword 'function'*, kemudian diikuti dengan nama *function* tersebut, parameter yang dituliskan di dalam kurung, dan *statement*/perintah *Javascript* yang ditulis di dalam kurung kurawal. Parameter pada *function* bisa lebih dari 1 yang penulisanya dipisahkan oleh tanda koma (,). *Function* bisa memiliki parameter atau tidak. Di bawah ini adalah potongan kode untuk membuat *function* penjumlahan 2 buah bilangan.

```
function penjumlahan(angka1,angka2){
    var hasil = angka1+angka2;
    return hasil;
}
```

Listing 3: *Function* penjumlahan 2 buah bilangan

Setelah membuat *function*, *function* tersebut tidak langsung dieksekusi. Membuat *function* hanya memberi nama *function* tersebut dan mendeskripsikan apa yang akan dilakukan oleh *function* tersebut apabila dipanggil. Dengan memanggil *function*, maka *function* akan dieksekusi. Di bawah ini adalah potongan kode untuk memanggil *function* dengan nama penjumlahan.

```
penjumlahan(10,5);
```

Listing 4: Memanggil *function* penjumlahan

Menggambar pada *Canvas*

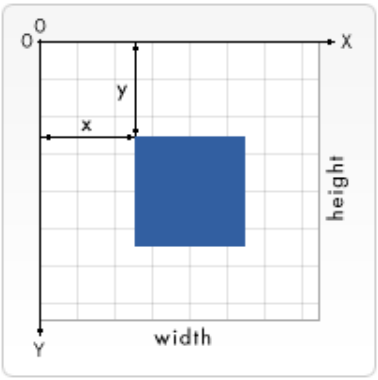
Sesudah menuliskan *tag* <canvas> pada HTML, canvas tidak bisa langsung digambar. Karena itu perlu ditambahkan *drawing context* pada *Javascript*. Di bawah ini adalah potongan kode untuk menambahkan *drawing context*.

```
var myCanvas = document.getElementById('canvas');
var context = myCanvas.getContext('2d');
```

Listing 5: Menambahkan *drawing context canvas*

Berdasarkan potongan kode di atas, variabel myCanvas menyimpan objek dengan id = 'canvas'. Id ini mengacu ke objek *canvas* pada HTML yang memilki id bernama canvas. Variabel myCanvas sekarang sudah menyimpan objek *canvas*. Kemudian variabel context menyimpan *drawing context* 2D. Sesudah itu, *canvas* tersebut dapat digambar dengan bentuk 2D, garis, kurva, membuat tulisan, dan menambahkan gambar. Selain untuk menggambar, bentuk-bentuk tersebut dapat diberi warna sesuai dengan keinginan.

Untuk menggambar bentuk 2D atau garis, diperlukan koordinat x dan y. Koordinat tersebut akan menempatkan gambar tersebut pada *canvas*. Posisi awal/*origin* pada *canvas* adalah (0,0) yang terletak di ujung kiri atas *canvas*. Gambar 1 adalah penempatan kotak biru pada *canvas* terhadap *origin*.



Gambar 1: Posisi kotak biru pada *canvas* terhadap *origin*

Pada Gambar 1, titik ujung kiri kotak biru tersebut berjarak x *pixel* dari kiri dan berjarak y *pixel* dari atas.

Menggambar Persegi Panjang

Ada 3 cara untuk menggambar persegi panjang:

- *fillRect(x,y,width,height)* : menggambar persegi panjang serta mengisi bagian tengah persegi panjang dengan warna.
- *strokeRect(x,y,width,height)* : menggambar *outline* yang berbentuk persegi panjang.
- *clearRect(x,y,width,height)* : menghapus daerah yang ditentukan pada *canvas*. Daerah yang dihapus berbentuk persegi panjang.
- *rect(x,y,width,height)* : menambah *path* berbentuk persegi panjang.

Fungsi tersebut memiliki parameter yang sama. Parameter x dan y untuk menentukan posisi pada canvas dari titik ujung kiri atas persegi panjang. *Width* adalah lebar dari persegi panjang dan *height* adalah tinggi dari persegi panjang.

Menggambar *Path*

Path adalah sekumpulan titik yang dihubungkan oleh segmen garis. *Path* dapat membentuk kurva dan membuat bentuk 2D lainnya seperti segitiga, trapesium, belah ketupat dan lain-lain. Langkah-langkah untuk membuat bentuk menggunakan path adalah sebagai berikut :

(a) Buat *path*.

(b) Tuliskan perintah untuk menggambar pada *path* tersebut.

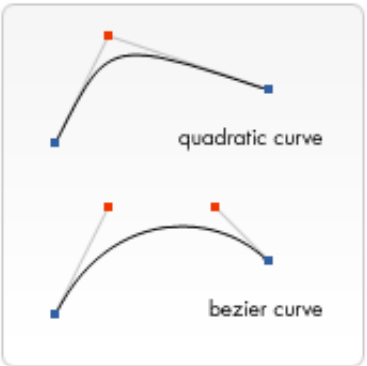
(c) Sesudah *path* tersebut sudah dibuat, *path* tersebut dapat di*render* menggunakan *stroke* atau *fill*.

Langkah pertama untuk membuat *path* baru adalah dengan menggunakan fungsi *beginPath()*. Setelah itu, perintah-perintah untuk menggambar dapat digunakan untuk membuat bentuk-bentuk yang diinginkan. Apabila sudah selesai menggambar, gunakan fungsi *stroke()* untuk menggambar outline dari *path* tersebut atau *fill()* untuk mengisi *area path* tersebut. Setelah itu, gunakan fungsi *closePath()* untuk menutup bentuk tersebut dengan cara menggambar garis lurus dari posisi titik terakhir ke titik awal.

Fungsi lainnya yang menjadi bagian dari membuat *path* adalah fungsi *moveTo()*. Fungsi ini diibaratkan seperti mengangkat sebuah pensil dari sebuah titik pada kertas kemudian menempatkanya pada titik yang diinginkan. Fungsi *moveTo()* memiliki 2 parameter yaitu x dan y yang merupakan posisi titik pada *canvas*. Ketika *canvas* sudah diinisialsasi dan fungsi *beginPath()* sudah dipanggil, fungsi *moveTo()* berguna sebagai penempatan titik awal untuk menggambar. Fungsi *lineTo()* memiliki 2 parameter yaitu x dan y yang merupakan titik akhir dari garis. Garis akan digambar mulai dari posisi titik awal sampai ke posisi titik akhir garis. Titik awal ini bergantung pada titik akhir dari *path* sebelumya. Titik awal dapat diubah dengan menggunakan fungsi *moveTo()*.

Fungsi *arc()* digunakan untuk menggambar lingkaran atau busur. Fungsi ini memiliki 6 parameter yaitu x, y, radius, *startAngle*, *endAngle* dan *anticlockwise*. Parameter x dan y adalah posisi titik tengah busur pada *canvas*. Radius adalah besar jari-jari busur. *StartAngle* dan *endAngle* adalah titik awal dan titik akhir busur dalam satuan radian yang diukur dari sumbu x. *Anticlockwise* adalah parameter yang bernilai *boolean*, apabila bernilai *true*, maka busur akan digambar berlawanan arah jarum jam dan jika bernilai *false*, busur akan digambar searah jarum jam. Karena fungsi *arc()* menerima input sudut dalam radian, maka perlu dilakukan konversi dari satuan derajat menjadi radian terlebih dahulu. Rumusnya adalah sebagai berikut : **radian = (Math.PI / 180) * besar sudut**

Bézier curve merupakan tipe *path* yang digunakan untuk membuat kurva. *Bézier curve* ada 2 jenis yaitu *cubic* dan *quadratic*. Perbedaanya adalah *quadratic Bézier curve* memiliki sebuah *control point*, sedangkan *cubic Bézier curve* memiliki 2 buah *control point*. Pada Gambar 2 menunjukkan perbedaan antara *quadratic Bézier curve* dan *cubic Bézier curve*. Titik merah pada gambar merupakan *control point* dari *Bézier curve*.



Gambar 2: Perbedaan *quadratic Bézier curve* dan *cubic Bézier curve*

Berikut adalah fungsi *quadratic* dan *cubic Bézier curve* :

- *quadraticCurveTo(cp1,cp2,x,y)* : menggambar *quadratic Bézier curve* dari posisi pensil sekarang ke titik akhir yaitu x dan y, dengan titik control point yaitu cp1 dan cp2.
- bezierCurveTo(cp1x,cp1y,cp2x,cp2y,x,y) : menggambar *cubic Bézier curve* dari posisi pensil sekarang ke titik akhir yaitu x dan y, dengan 2 titik control point yaitu (cp1x,cp1y) dan (cp2x,cp2y).

Object Oriented Programming Javascript

OOP (*Object Oriented Programming*) adalah sebuah paradigma *programming* yang menggunakan abstraksi untuk membuat objek-objek yang ada pada dunia nyata. Bahasa pemrograman seperti *Java*, C++, *Ruby, Phyton*, PHP, dan *Objective-C* sudah mendukung OOP. Dalam OOP, setiap objek dapat menerima pesan, memproses data dan mengirim pesan ke objek lain. Program yang menggunakan konsep OOP ini mudah untuk dimengerti dan lebih mudah untuk dikembangkan oleh *programmer*.

Ide umum pada OOP adalah menggunakan objek untuk memodelkan benda-benda yang ada pada dunia nyata. Objek tersebut kemudian direpresentasi pada program yang dibuat. Objek-objek dapat berisi data, fungsionalitas dan *behaviour* yang merepresentasikan informasi tentang objek tersebut dan tugas objek. Contohnya, membuat objek sebuah mobil. Mobil memiliki beberapa informasi diantaranya adalah merk mobil, berat mobil, warna mobil dan tahun produksi. Informasi tersebut dapat disebut sebagai properti dari objek. Mobil dapat bergerak maju, berbelok ke kanan, berbelok ke kiri, bergerak mundur dan berhenti. Hal-hal yang dapat dilakukan oleh objek disebut sebagai method dari objek.

Kelas

Javascript tidak memiliki *statement 'class'* yang dapat digunakan pada bahasa pemrograman C++ atau *Java*. Untuk membuat kelas, *Javascript* menggunakan *function* sebagai konstruktor untuk kelas. Karena itu, membuat kelas sama dengan membuat *function* pada *Javascript*. Di bawah ini adalah potongan kode untuk membuat kelas bernama Mobil.

```
1    function Mobil(){}
```

Listing 6: Membuat kelas Mobil

Objek

Untuk membuat instansi baru dari objek, gunakan *statement 'new'* yang nantinya akan disimpan pada variabel. Di bawah ini adalah potongan kode untuk membuat instansi.

```
1    var mobil1 = new Mobil();
```

Listing 7: Membuat *instance* mobil

Konstruktor

Konstruktor adalah *method* yang ada pada kelas. Konstruktor akan dipanggil ketika pertama kali inisialisasi atau saat instansi dari objek baru dibuat. *Function* pada *Javascript* berfungsi sebagai konstruktor sehingga tidak perlu membuat method konstruktor lagi. Semua aksi yang terdapat pada kelas akan dieksekusi pada saat instansiasi.

Properti/Atribut

Properti adalah variabel yang terdapat pada kelas. Properti ditulis pada konstruktor kelas sehingga setiap properti pada kelas akan dibuat ketika membuat instansi baru. Untuk membuat

properti, gunakan *statement 'this'*. *Statement 'this'* juga dapat diartikan bahwa atribut-atribut yang dibuat merupakan atribut milik objek. Cara ini mirip dengan bahasa pemrograman *Java* ketika membuat sebuah properti pada objek. Sintaks untuk mengakses properti di luar kelas adalah : namaInstansi.properti. Di bawah ini adalah potongan kode untuk mendefinisikan properti pada kelas Mobil pada saat instansiasi.

```
function Mobil(merkMobil,beratMobil,warnaMobil,tahunProduksi){
    this.merkMobil = merkMobil;
    this.beratMobil = beratMobil; //satuan dalam kg
    this.warnaMobil = warnaMobil;
    this.tahunProduksi = tahunProduksi;
}

var mobil1 = new Mobil('Toyota',1000,'Hitam',2010);
```

Listing 8: Mendefinisikan properti pada kelas Mobil

Method

Method adalah hal yang dapat dilakukan oleh sebuah objek. Untuk membuat *method*, tuliskan nama *method* terlebih dahulu kemudian *assign* fungsi pada nama method tersebut. Untuk memanggil *method* sebuah objek, tuliskan nama objek/kelas terlebih dahulu, kemudian tuliskan nama *method* sesuai dengan yang sudah dibuat beserta tanda kurung. Tanda kurung berisi parameter. Di bawah ini adalah potongan kode untuk membuat dan memanggil method bergerakMaju() pada kelas Mobil.

```
function Mobil(merkMobil,beratMobil,warnaMobil,tahunProduksi){
    this.merkMobil = merkMobil;
    this.beratMobil = beratMobil; //satuan dalam kg
    this.warnaMobil = warnaMobil;
    this.tahunProduksi = tahunProduksi;

    this.bergerakMaju = function(){
        //kode agar mobil bergerak maju
    }
}

var mobil1 = new Mobil('Toyota',1000,'Hitam',2010);
mobil1.bergerakMaju(); //memanggil fungsi untuk bergerak maju
```

Listing 9: Membuat dan memanggil method bergerakMaju()

Event

Event adalah kejadian/peristiwa yang terjadi pada sistem yang diprogram. Sistem akan memberitahu apabila kejadian tersebut sudah terjadi dan akan melakukan suatu aksi ketika kejadian sudah terjadi. Misalnya, di bandara ketika landasan pacu sudah bersih untuk pesawat lepas landas, sinyal akan dikomunikasikan kepada pilot bahwa pesawat sudah boleh untuk lepas landas. Dalam *web*, *event* ditembakan di dalam *browser window* dan dikaitkan pada objek yang spesifik seperti sekumpulan elemen, dokumen HTML yang dimuat atau keseluruhan *browser window*. Ada beberapa *event* yang dapat terjadi diantaranya adalah :

- Pengguna mengklik sebuah element atau mengarahkan kursor ke sebuah elemen.
- Pengguna menekan sebuah tombol pada *keyboard*.

- Pengguna mengatur besar dan menutup *browser window*.
- Halaman *web* selesai dimuat.
- *Form* sedang di*submit*.
- Video sedang dimainkan, dijeda, atau selesai.
- Ketika *error* terjadi.

Setiap *event* memiliki *event handler*, yang berisikan sekumpulan kode yang akan dijalankan ketika *event* sudah terjadi. *Event handler* juga sering disebut sebagai *event listener*. *Listener* menunggu *event* yang terjadi dan *handler* adalah kode yang dijalankan ketika *listener* mendapatkan *event*/ketika *event* terjadi. Untuk memperjelas bagaimana cara menggunakan *event*, di bawah ini terdapat contoh kode untuk menambahkan event pada button/tombol.

```
<html>
    <title>Event pada tombol</title>
    <body>
        <button id='tombol'>Change color</button>
    </body>
</html>

<script>
    var btn = document.getElementById('tombol');

    function random(number) {
        return Math.floor(Math.random()*(number+1));
    }

    btn.onclick = function() {
        var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' +
            random(255) + ')';
        document.body.style.backgroundColor = rndCol;
    }
</script>
```

Listing 10: Menambahkan event pada button

Berdasarkan kode di atas, objek *button* dengan id='tombol' disimpan di dalam variabel bernama 'btn'. Ada fungsi bernama *'random'* untuk mengembalikan sebuah nilai acak. Setelah itu ada *event handler*. *Event handler property* yang digunakan adalah *onclick*. *Event handler property onclick* mengecek apakah objek(dalam kasus ini objeknya adalah button) sudah ditekan/diklik. Bila tombol sudsh diklik, maka akan mengeksekusi fungsi untuk mengubah warna *background*. Warna RGB tersebut di*generate* secara acak menggunakan fungsi *random* yang sudah dibuat sebelumnya. Tidak hanya *event handler property onclick* saja yang dapat digunakan pada halaman web. Berikut ini adalah beberapa *event handler property* lainnya:

- *onfocus* dan *onblur* : event akan terjadi apabila sebuah objek difokuskan/tidak. Biasanya digunakan untuk menampilkan informasi tentang bagaimana cara mengisi *form* ketika difokuskan atau menampilkan pesan *error* ketika *form* tersebut diisi dengan nilai yang salah/tidak valid.
- *ondblclick* : *event* akan terjadi ketika objek diklik 2 kali/*double click*.
- *window.keypress, window.onkeydown, window.onkeyup* : *event* akan terjadi apabila sebuah tombol pada *keyboard* ditekan. *Keypress* adalah *event* ketika tombol ditekan kemudian dilepas. *Keydown* adalah *event* ketika tombol ditekan dan *keyup* adalah *event* ketika tombol

dalam keadaan tidak ditekan. Untuk ketiga *event* ini, event tersebut harus di*register* pada objek *window* yang merepresentasikan *browser window*.

– *onmouseover* dan *onmouseout* : event akan terjadi ketika posisi kursor *mouse* berada luar objek lalu ditempatkan di atas objek dan ketika posisi kursor *mouse* berada di atas objek lalu keluar dari objek.

Beberapa *event handler property* tersebut sangat umum dan tersedia di manapun, sedangkan beberapa *event handler property* lainnya sangat spesifik dan hanya digunakan untuk elemen tertentu, contohnya adalah menggunakan *onplay* untuk elemen tertentu yaitu <video>.

Mekanisme event terbaru dalam spesifikasi DOM(*Document Object Model*) *level* 2 *Events* yang memberikan *browser* sebuah fungsi baru yaitu *addEventListener()*. Fungsi ini mirip seperti *event handler property* namun memiliki sintaks yang berbeda. Di bawah ini adalah potongan kode untuk menggunakan fungsi *addEventListener()*.

```
1  var btn = document.getElementById('tombol');
2
3  function bgChange() {
4      var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random
           (255) + ')';
5      document.body.style.backgroundColor = rndCol;
6  }
7
8  btn.addEventListener('click', bgChange);
```

Listing 11: Menggunakan fungsi addEventListener()

Pada fungsi *addEventListener()*, ada 2 buah parameter yaitu *event* yang ingin digunakan(dalam potongan kode di atas menggunakan *event click*) dan kode sebagai *handler* yang ingin dijalankan ketika *event* tersebut terjadi. Selain cara di atas, dapat juga menuliskan semua kode di dalam fungsi *addEventListener()* seperti potongan kode di bawah ini.

```
1  btn.addEventListener('click', function() {
2      var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random
           (255) + ')';
3      document.body.style.backgroundColor = rndCol;
4  });
```

Listing 12: Menuliskan kode di dalam fungsi addEventListener()

Membuat Animasi

Langkah-langkah untuk membuat animasi adalah sebagai berikut :

(a) Membersihkan *canvas* : hilangkan semua bentuk-bentuk yang sudah tergambar di canvas. Untuk menghapus keseluruhan *canvas*, gunakan fungsi *clearRect()*.

(b) Menyimpan *state canvas* : ketika mengubah atribut(seperti *style*) yang mempengaruhi *state canvas* dan ingin *original state* tersebut digunakan kembali, *state* tersebut harus disimpan.

(c) Gambar bentuk : gambar bentuk yang ingin dianimasikan.

(d) Mengembalikan *state canvas* : jika state sudah disimpan, kembalikan *state* tersebut sebelum menggambar di *frame* yang baru.

Bentuk yang digambar pada *canvas* dapat menggunakan fungsi yang dimiliki oleh *canvas* atau dengan membuat fungsi sendiri. Hasil yang ada pada *canvas* akan muncul setelah *script* selesai dieksekusi. Jadi dibutuhkan cara mengeksekusi fungsi secara terus menerus untuk menggambar dalam waktu tertentu. Ada 3 fungsi yang dapat digunakan untuk memanggil fungsi dalam kurun waktu tertentu diantaranya adalah :

- *setInterval(function,delay)*: mengeksekusi fungsi *function* berulang kali setiap *delay* milidetik.
- *setTimeout(function,delay)*: mengeksekusi fungsi *function* setiap *delay* milidetik.
- *requestAnimationFrame(callback)*: memberitahu *browser* untuk menjalankan animasi dan meminta *browser* memanggil fungsi yang spesifik untuk memperbarui animasi.

Jika tidak ingin ada iteraksi user, gunakan fungsi *setInterval()* untuk mengeksekusi fungsi berulang kali. Bila ingin ada interaksi user, terutama dalam pembuatan game yang membutuhkan input *keyboard* atau *mouse* untuk mengontrol animasi, gunakan fungsi setTimeout().

- ***jQuery***

jQuery merupakan *library Javascript*. Semua yang ada pada *jQuery* dapat diakses melalui *Javascript*. Untuk dapat menggunakan *jQuery*, perlu menambahkan *script jQuery* pada tag <script> di bagian *src*. Untuk memanggil objek menggunakan *jQuery* harus diawali simbol $, kemudian diikuti dengan objek dan *method*nya. Contoh penulisan pada *jQuery* adalah sebagai berikut : *$("h1").remove()*.

Sebuah halaman *web* tidak dapat dimanipulasi dengan aman apabila halaman tersebut belum siap. *jQuery* menyediakan *$(document).ready()* untuk mengecek apakah halaman sudah siap. Kode yang berada di dalam *$(document).ready()* akan dijalankan oleh *Javascript* setelah halaman DOM(*Document Object Model*) sudah siap dimuat. Berikut adalah potongan kode untuk menuliskan *$(document).ready()*:

```
1    $( document ).ready(function() {
2        //tuliskan kode di bagian ini
3    });
```

Listing 13: Menuliskan method *$(document).ready()*

Method .attr() berfungsi sebagai *setter* dan *getter*. Sebagai *setter*, *method .attr()* dapat menerima sebuah *key* dan *value* atau sebuah objek yang memiliki satu atau lebih *key/value pair*. Di bawah ini adalah contoh *method .attr()* sebagai *setter* dan *getter*.

```
1    $( "a" ).attr( "href", "allMyHrefsAreTheSameNow.html" );
2
3    $( "a" ).attr({
4    title: "all titles are the same too!",
5    href: "somethingNew.html"
6});
```

Listing 14: Method .attr() sebagai setter

```
1    $( "a" ).attr( "href" ); //mengembalikan href yang memiliki elemen a
         pertama pada dokumen
```

```
});
```

Listing 15: Method .attr() sebagai getter

Konsep yang paling umum pada *jQuery* adalah menyeleksi beberapa elemen dan melakukan sesuatu terhadap elemen tersebut. Ada beberapa cara untuk menyeleksi beberapa elemen diantaranya adalah sebagai berikut:

- *$("#myId")* : mendapatkan elemen berdasarkan id. Id harus unik per halaman.
- *$(".myClass")* : mendapatkan elemen berdasarkan nama kelas.
- *$("input[name='first_name']")* : mendapatkan elemen berdasarkan atributnya.
- *$("#contents ul.people li")* : mendapatkan elemen berdasarkan *Compound CSS Selector*.
- *$("div.myClass, ul.people")* : mendapatkan elemen berdasarkan sekumpulan *selector*.

Untuk mengecek apakah elemen tidak kosong, maka cek elemen tesebut dengan menggunakan *method .length()*. Apabila nilai tersebut 0, maka elemenya kosong. *jQuery* tidak menyimpan elemen. Untuk itu dibutuhkan variabel untuk menyimpan elemen tersebut jika ingin mendapatkan elemen lain. Di bawah ini adalah potongan kode untuk menyimpan elemen pada variabel.

```
    var divs = $( "div" );
});
```

Listing 16: Menyimpan elemen pada variabel

Beberapa *method* pada *jQuery* dapat digunakan untuk mendapatkan atau me-*assign* nilai pada saat seleksi. Ketika *method* dipanggil dengan sebuah nilai sebagai argumen, *method* tersebut dapat disebut sebagai *setter*. Ketika method dipanggil tanpa argumen, method tersebut mendapatkan nilai dari elemen/sebagai *getter*.

```
    $( "h1" ).html( "hello world" ); //assign elemen h1 dengan 'hello world'
    $( "h1" ).html(); // mendapatkan nilai elemen h1 pertama
});
```

Listing 17: Setter dan getter

Ada beberapa cara untuk mengubah elemen yang ada. Cara yang paling umum adalah mengubah *inner HTML* atau atribut dari sebuah elemen. *jQuery* menyediakan *method-method* untuk mengubah/memanipulasinya. Cara ini juga dapat digunakan untuk mendapatkan informasi tentang elemen tersebut. Di bawah ini adalah beberapa method yang digunakan untuk mendapatkan dan mengubah informasi dari elemen :

- *.html()* : mendapatkan atau mengubah konten HTML.
- *.text()* : mendapatkan atau mengubah konten tulisan.
- *.attr()* : mendaptakan atau mengubah nilai atribut yang diinginkan
- *.width()* : mendapatkan atau mengubah lebar sebuah elemen.
- *.height()* : mendapatkan atau mengubah tinggi sebuah elemen.
- *.position()* : mendapatkan posisi sebuah objek. Method ini tidak dapat digunakan untuk mengubah nilai.
- *.val()* : mendapatkan atau mengubah nilai dari elemen pada *form*.

Events

jQuery dapat mengatur *event* pada halaman *web*. *Event* ini biasanya akan terjadi apabila ada

interaksi pengguna dengan halaman *web*, misalnya ketika mengisi teks pada *form* atau kursor pada mouse dipindahkan. *jQuery* menyediakan *method event* untuk *native browser*, diantaranya adalah *.click(), .focus(), .blur(), .change()* dan masih banyak lagi. Selain itu method *.on()* berguna untuk *binding* fungsi *handler* yang sama untuk banyak *event*. Di bawah ini adalah contoh potongan kode *event jQuery*.

```
1    $( "p" ).click(function() {
2        console.log( "You clicked a paragraph!" );
3    });
```

Listing 18: Event click

```
1    $( "p" ).on( "click", function() {
2        console.log( "click" );
3    });
```

Listing 19: Event click menggunakan method .on()

Setiap *event handling* menerima *object event* yang berisi banyak properti dan *method*. *Event object* mengandung beberapa properti dan *method* yang berguna, diantaranya adalah:

- *pageY, pageX* : posisi kursor *mouse* ketika *event* terjadi. Titik (0,0) berada pada sudit kiri atas *browser window*.
- *type* : tipe *event*(contohnya : *click*).
- *which* : tombol yang ditekan.
- data : data yang diberikan ketika *event* terjadi.
- target : elemen DOM yang memulai *event*.
- *timeStamp* : perbedaan dalam milisekon antara *event* terjadi pada *browser* dengan tanggal 1 Januari 1970.
- *stopPropagation()* : menghentikan *event* dari *bubbling* dengan elemen lain.

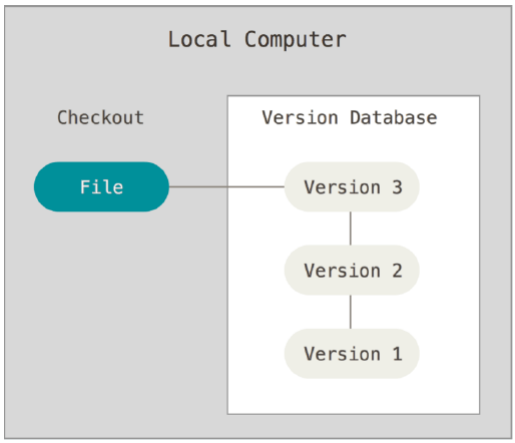
- ***Git***

  Version Control

  Version control adalah sistem yang menyimpan perubahan pada sebuah *file* atau sekumpulan *file* secara berkala sehingga dapat mendapatkan versi yang spesifik nantinya. VCS(*Version Control System*) memungkinkan pengguna untuk mengembalikan *file* yang diinginkan ke *state* sebelumnya, mengembalikan keseluruhan proyek ke *state* sebelumnya, membandingkan perubahan secara berkala, dapat melihat pengguna terakhir yang memodifikasi sesuatu yang menyebabkan masalah, dan masih banyak lagi. Ketika beberapa *file* ada yang hilang karena sebuah kesalahan, *file-file* tersebut dapat dikembalikan dengan mudah.

  Local Version Control System

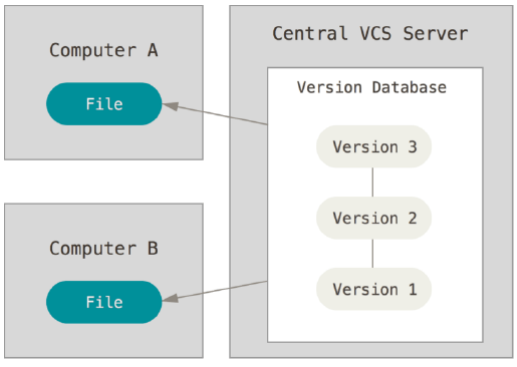
  Local Version Control System memiliki sebuah basis data yang menyimpan semua perubahan pada *file* dalam *revision control*. Salah satu VCS tools yang cukup terkenal adalah RCS yang masih digunakan oleh banyak komputer hingga sekarang. Cara kerja RCS adalah menyimpan *patch sets* yang merupakan perbedaan antara beberapa *file* seperti pada Gambar 3. *Patch sets* tersebut disimpan di *disk*. RCS dapat menampilkan *file* apa saja pada suatu waktu dengan menggabungkan *patch-patch* tersebut.



Gambar 3: Local Version Control

Centralized Version Control System

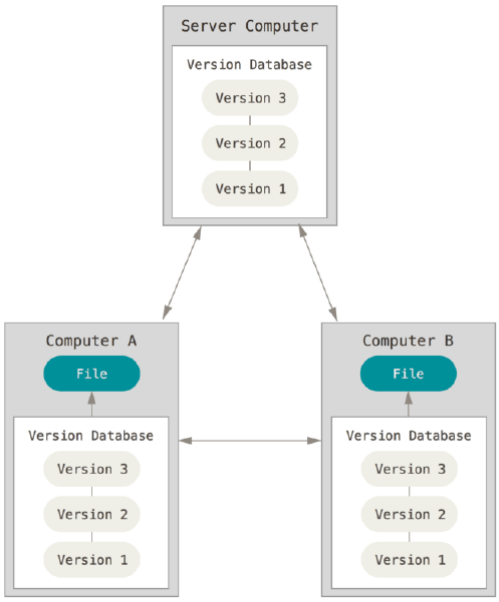
CVCS(*Centralized Version Control System*) memiliki sebuah *server* yang menyimpan semua *file* beserta historynya dan jumlah *client* yang mengecek *file* tersebut. Dengan adanya CVCS, semua orang mengetahui apa yang dilakukan oleh kolaborator yang mengerjakan proyek. Tetapi kelemahanya adalah ketika *server* tersebut *down*, tidak akan ada yang bisa berkolaborasi dan tidak dapat menyimpan perubahan yang sudah dikerjakan. Selain itu apabila data di *server* tersebut hilang maka dan tidak melakukan *back-up*, proyek yang sedang dikerjakan akan hilang beserta semua historinya. Struktur CVCS dapat dilihat pada Gambar 4.



Gambar 4: Centralized Version Control

Distributed Version Control System

Dalam DVCS(*Distributed Version Control System*) seperti *Git, Mercurial, Bazaar dan Darcs*, *client* tidak mengecek versi terbaru dari *file* tetapi *client* menggandakan *repository* termasuk historinya. Jika *server* mati/kehilangan data, maka *client* memiliki *file back-up* untuk mengembalikanya. Ilustrasi DVCS terdapat pada Gambar 5.



Gambar 5: Distributed Version Control

Git

Git merupakan sebuah *version control* namun berbeda dengan VCS lainnya dilihat dari cara menyimpan datanya. Sistem seperti CVS, *Subversion, Perforce, Bazzar* menyimpan data sebagai sekumpulan *file* dan perubahan setiap *file* disimpan setiap waktu. Pada *Git*, data tersebut dianggap sebagai sekumpulan *snapshot* dari *miniature filesystem*. Setiap *commit* atau menyimpan proyek, *Git* seolah-olah mengambil gambar untuk melihat seperti apa *file* yang terlihat pada saat itu dan menyimpannya sebagai referensi pada *snapshot* tersebut. Singkatnya, apabila tidak ada *file* yang diubah, Git tidak akan menyimpan *file* lagi.

Hampir semua operasi pada *Git* dapat dilakukan secara lokal. Ketika ingin menlihat histori suatu proyek, *Git* akan mengambil data histori tersebut dari basis data lokal, sehingga tidak perlu memintanya ke *server*. Selain itu, pengguna dapat bekerja secara *offline*. Pada sistem lain seperti *Perforce*, pengguna tidak dapat melakukan banyak hal jika tidak terkoneksi ke *server* dan pada CVS, pengguna dapat mengubah *file* tetapi tidak dapat *commit* ke basis data. Pada *Git*, pengguna dapat *commit* dikarenakan *Git* memiliki basis data lokal.

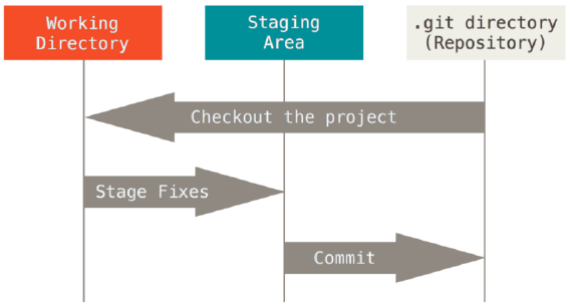
Git memiliki 3 *state* utama pada *file* yaitu:

- *committed* : data sudah tersimpan di basis data lokal.
- *modified* : *file* sudah diubah namun belum di*commit* ke basis data.
- *staged* : menandai file yang sudah dimodifikasi dalam versi sekarang untuk di*commit*.

Terdapat 3 bagian utama dalam proyek *Git* yaitu :

- *Git directory* : tempat untuk menyimpan *metadata* dan objek basis data untuk proyek yang dibuat. Ini adalah bagian terpenting dari *Git* dan inilah yang di-*copy* ketika *clone repository* dari komputer lain.
- *Working tree* : *single checkout* sebuah versi dari proyek. *File* diambil dari basis data yang sudah di*compressed* di *Git directory* dan disimpan pada *disk* untuk digunakan dan dimodifikasi.
- *Staging area* : sebuah *file* yang ada di *Git directory* yang menyimpan informasi tentang apa yang akan disimpan untuk *commit* selanjutnya.

Gambar 6 di bawah ini menunjukan *working tree, staging area dan Git directory.*



Gambar 6: Working tree, staging area, dan Git directory

Workflow pada *Git* adalah sebagai berikut :

(a) Pengguna memodifikasi *file* di *working tree* milik pengguna.

(b) Pengguna memilih *file* yang akan menjadi bagian dari *commit* selanjutnya. *File* yang terpilih akan ditambahkan ke *staging area.*

(c) Pengguna melakukan *commit file* tersebut yang berada pada *staging area* dan menyimpan *snapshot* secara permanen ke *Git directory.*

Apabila versi tertentu dari sebuah *file* sudah ada pada *Git directory*, maka *file* tersebut dalam berada dalam *state committed.* Jika *file* sudah dimodifikasi dan sudah ditambahkan ke *staging area*, maka file tersebut dalam state *staged.* Jika *file* sudah diubah dan sudah di*checkout* tetapi belum dalam *state staged*, maka *file* tersebut dalam *state modified.*

Ada beberapa cara dalam menggunakan *Git* yaitu dengan menggunakan *command-line* dan beberapa GUI(*Graphical User Interface*) yang memiliki kemampuan bermacam-macam. Pada umumnya digunakan *command-line*, karena *command-line* dapat menjalankan semua perintah *Git* sedangkan GUI hanya memiliki sebagian fungsionalitas pada *Git* supaya mudah digunakan.

Mendapatkan *Git Repository*

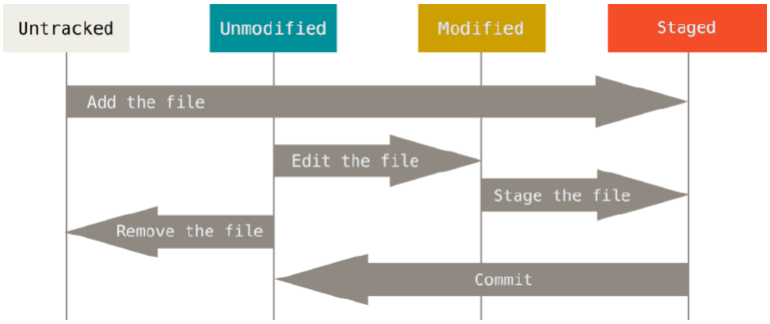
Untuk mendapatkan *Git repository* ada 2 cara yaitu : menjadikan sebuah proyek yang terdapat pada direktori lokal yang belum dalam *version control* lalu menjadikannya sebagai *Git repository* atau *clone Git repository* yang sudah ada.

Jika memiliki direktori proyek yang belum dalam *version control* dan ingin mengontrolnya menggunakan *Git*, hal pertama yang harus dilakukan adalah dengan membuka direktori proyek. Perintah untuk membuat repository pada *Windows* adalah dengan mengetikan perintah *cd /c/user/my_project* sesudah itu ketik perintah *git init.* Perintah tersebut akan membuat subdirektori bernama *.git* yang mengandung semua repository yang dibutuhkan. Setelah mengetikan perintah di atas, proyek tersebut belum di-*track* sama sekali. Untuk men-*track file-file* pada sebuah proyek, pertama gunakan perintah *git add* untuk men-*track file* yang diinginkan kemudian ketik *git commit* untuk commit file tersebut.

Clone repository adalah mendapatkan *copy* dari *repository* yang sudah ada. Perintah yang digunakan adalah *git clone.* Tidak hanya *file-file* pada *repository* saja yang di*copy*, tetapi semua histori pada *repository* tersebut akan ikut ter*copy.* Perintah *git clone* diikuti dengan *url. Url* ini berisi *link* di mana *repository* berada.

Record* Perubahan pada *Repository

Setiap *file* dalam direktori memiliki 2 *state* yaitu *tracked* atau *untracked*. *Tracked file* adalah *file* yang berada pada *snapshot* terakhir. *Tracked file* adalah *file* yang *Git* ketahui sekarang. *Untracked file* adalah *file* yang tidak berada pada *snapshot* terakhir. Ketika *file* diubah, *Git* melihat bahwa *file* tersebut sudah dimodifikasi, karena *file* tersebut diubah setelah *commit* terakhir. Kemudian *file* yang sudah dimodifikasi tersebut di-*stage* dan *commit* semua *file* yang sudah di*staged* tersebut. Gambar 7 menunjukan siklus hidup dari status *file*.



Gambar 7: Siklus hidup pada status file

Perintah *git status* digunakan untuk mengecek status *file*. Jika mengetik perintah sesudah *clone*, maka tidak ada *untracked file* karena pada saat *clone*, tidak ada *file* yang dimodifikasi. Bila menambahkan sebuah *file* baru atau mengubah *file* lalu mengetik perintah *git status*, maka akan diberitahukan bahwa terdapat *untracked file*. Karena itu untuk men-*track file* baru, gunakan perintah *git add* yang diikuti dengan nama *file*nya seperti contoh ini : *git add README*. Perintah *git add* tidak hanya digunakan untuk men-*track file* baru. Selain digunakan untuk men-*track file*, perintah *git add* digunakan untuk stage *file* yang sudah dimodifikasi.

Tidak semua *file* akan ditambahkan secara otomatis oleh *Git* atau ada *file* yang ditunjukan sebagai *file untracked*. Hal ini dapat diatasi dengan membuat sebuah *file* yang bernama *.gitignore*. *File .gitignore* ini berisi *file-file* yang tidak akan di-*track* oleh *Git*. *File* yang biasanya ada dalam *.gitignore* adalah *log*, *tmp* atau *file* dokumentasi yang di*generate* secara otomatis. Adapun aturan untuk *pattern* yang dapat dimasukan pada *file .gitignore* diantaranya adalah :

- Baris kosong atau baris yang diawali dengan tanda pagar(#) akan dibiarkan.
- *Standard glob patterns*.
- *Pattern* diawali dengan garis miring(/) untuk mencegah rekusrif.
- *Pattern* diakhiri dengan garis miring untuk menspesifikasikan direktori.
- Menegasikan *pattern* diawali dengan tanda seru(!).

Glob pattern adalah *regular expression* yang digunakan oleh *shells*. Tanda bintang(*) untuk nol atau beberapa karakter, [abc] untuk karakter apa saja yang berada di dalam kurung siku, tanda tanya(?) untuk sebuah karakter apa saja dan tanda kurung siku dengan tanda strip(-) untuk karakter antara sebuah karakter dengan karakter lainya.

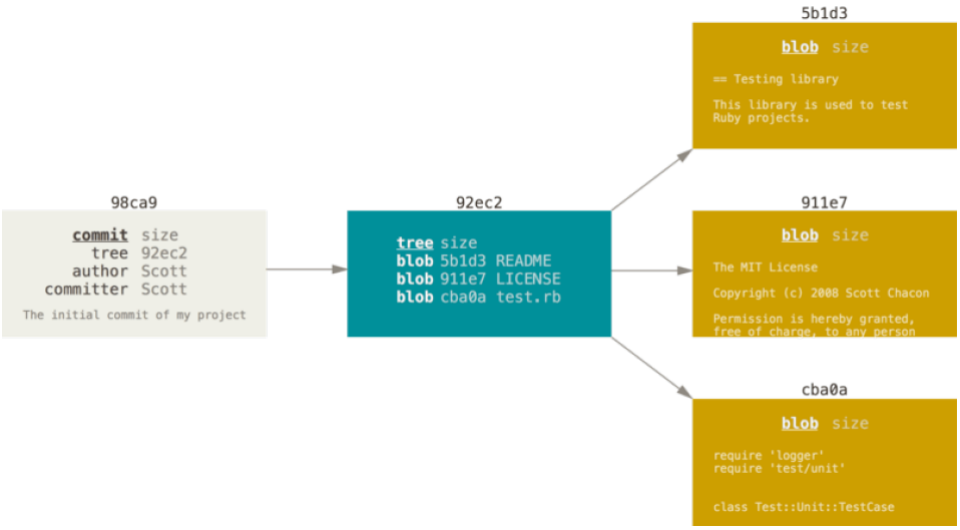
Perintah *git commit* digunakan untuk *commit file* yang sudah diubah dan ditambahkan. *File* tersebut harus sudah di-*stage* dengan menggunakan perintah *git add*. *File* yang belum di-*stage* akan berada dalam state *modified* meskipun sudah melakukan *commit*. Untuk menambahkan keterangan tentang *file* yang di*commit* dapat dituliskan perintah *git commit -m* yang diikuti dengan

keterangan yang ingin disampaikan.

Git Branching

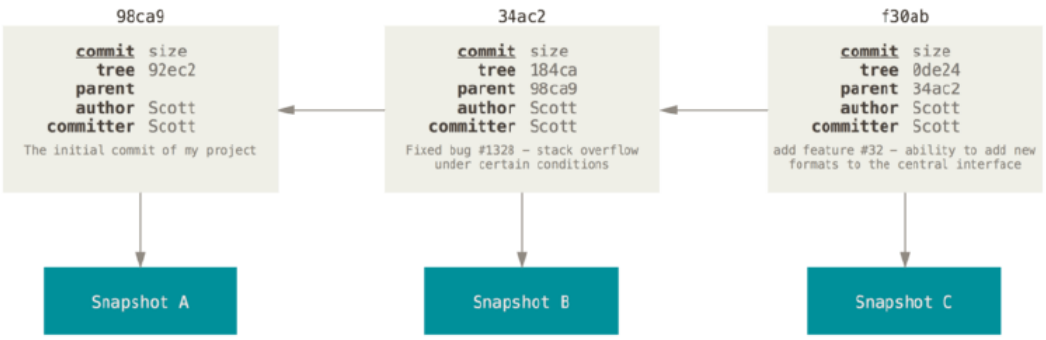
Branching artinya membuat dan mengerjakan sebuah proyek di tempat yang berbeda namun masih dalam repository yang sama sehingga tidak mengubah proyek utama. Ketika *commit*, *Git* menyimpan objek *commit* yang memiliki sebuah *pointer* pada *snapshot* sebuah konten yang sudah dalam *state staged*. Objek ini mengandung nama pembuat dan alamat email, pesan yang diketik, dan *pointer* ke *commit*.

Misalkan seorang pengguna memiliki 3 *file*, kemudian file tersebut semuanya di-*stage* dan *commit*. *Staging file* akan mengkomputasi *checksum* untuk setiap *file*, menyimpan versi tersebut pada *Git repository*(hal ini dapat disebut juga sebagai *blobs*), dan menambah *checksum* tersebut ke *staging area*. Lalu *Git* melakukan *checksum* pada setiap *subdirectory* dan menyimpan ketiga objek tersebut pada *Git repository*. Sesudah itu *Git* akan membuat objek *commit* yang mengandung *metadata* dan *pointer* ke proyek *root* sehingga dapat melihat *snapshot* tersebut pada setiap versi. Sekarang, *Git repository* memiliki 5 objek yaitu 3 *blob* yang merepresentasikan 3 file, sebuah *tree* yang mengandung isi direktori dan memberi nama *blob* berdasarkan nama file yang di*commit*, dan sebuah *commit* dengan *pointer* ke *root tree* dan semua *commit metadata*. Gambar 8 merupakan *tree* dari penjelasan tersebut.



Gambar 8: Commit dan tree dari file yang dicommit

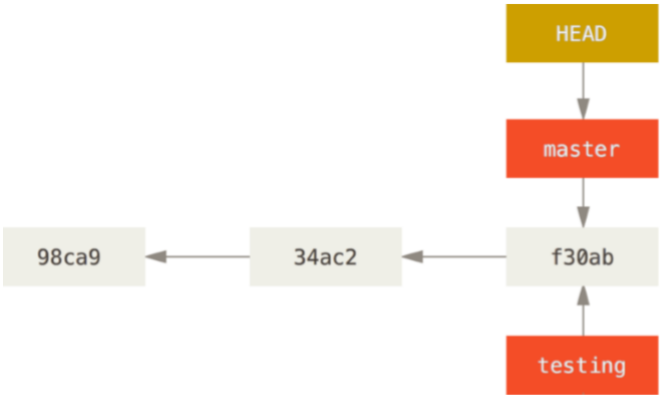
Jika ada perubahan pada proyek dan *commit* proyek tersebut, maka *commit* sesudahnya menyimpan *pointer* pada *commit* sebelum *commit* terbaru seperti yang terdapat pada Gambar 9. Jadi *parent* dari sebuah *commit* adalah *commit* sebelumnya dan kemudian seterusnya.



Gambar 9: Commit dan parent dari commit

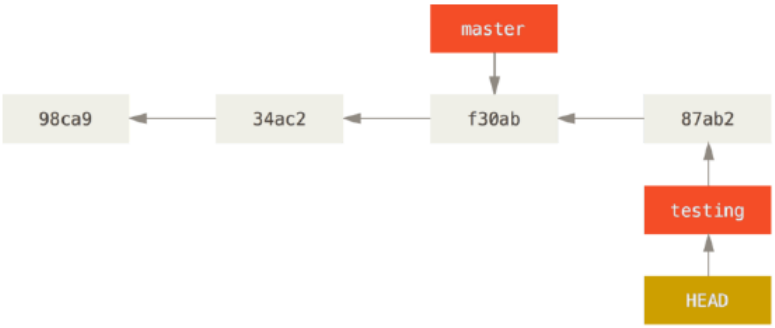
Nama *branch* pada *Git* awalnya disebut *master*. Ketika *commit*, pengguna diberikan *branch master* yang menunjuk pada *file* yang di*commit* terakhir. Setiap *commit*, pointer pada *branch master* akan terus maju secara otomatis.

Untuk membuat branch baru, gunakan perintah *git branch* diikuti dengan nama *branch*. *Git* menggunakan *pointer* yang disebut dengan *HEAD* untuk mengetahui bahwa pengguna sedang berada dalam *branch* tertentu. Bila membuat *branch* baru, posisi *HEAD* tetap berada pada *branch* yang sekarang. Perintah *git branch* hanya membuat *branch* baru dan tidak berpindah ke *branch* yang baru saja dibuat. Pada Gambar 10, jika mengetikan perintah *git branch testing*, *branch testing* akan dibuat tetapi *pointer HEAD* akan tetap berada pada *branch master*.



Gambar 10: *Pointer HEAD* menunjuk *branch master*

Untuk pindah *branch*, gunakan perintah *git checkout* diikuti dengan nama *branch*. *Pointer HEAD* akan berpindah ke *branch* tersebut. Bila pada *branch* tersebut pengguna melakukan *commit*, maka *branch* tersebut akan maju beserta dengan *pointer HEAD* seperti dicontohkan pada Gambar 11. Misalkan pengguna *commit* pada *branch testing*, maka hanya *branch testing* saja yang maju sedangkan *branch master* tidak. Ini dikarenakan *file* pada *branch master* tidak diubah.

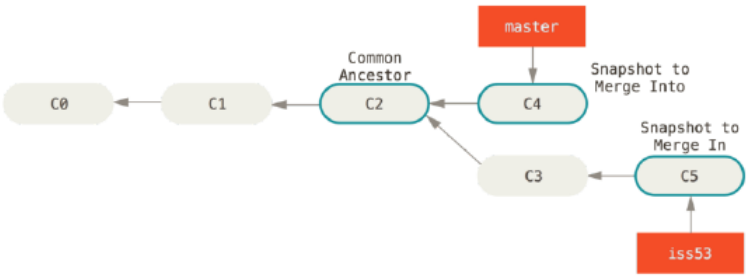


Gambar 11: *Pointer HEAD* beserta *branch testing*

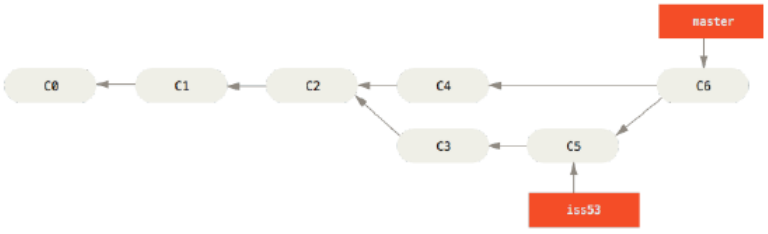
Perintah *git checkout* tidak hanya sebatas untuk pindah ke *branch* yang diinginkan. *File* yang ada pada *working directory* akan diubah dengan *file* yang ada pada *branch* tersebut. Bila berpindah ke *branch* sebelumnya, maka *file* dalam *working directory* akan dikembalikan sesuai dengan *commit* terakhir dari *branch* tersebut. Untuk membuat *branch* baru sekaligus pindah *branch*, gunakan perintah *git checkout -b* diikuti dengan nama *branch* yang ingin dibuat. Dengan ini *pointer HEAD* akan berada pada *branch* yang baru dibuat.

Basic Merging

Merging adalah penggabungan sebuah *branch* dengan *branch* lain. Perntah untuk *merge* adalah *git merge* diikuti dengan nama *branch* yang ingin digabungkan. Bila sebuah *branch* ingin digabungkan dengan *branch* yang memiliki *direct ancestor* yang berbeda, *Git* akan melakukan *three way merge*. *Three way merge* ini menggunakan 2 *snapshot* yang menunjuk pada *branch* yang akan digabungkan dan 1 *snapshot* yang menunjuk pada *ancestor* yang sama dari kedua *branch* tersebut seperti yang terdapat pada Gambar 12. Kemudian *Git* membuat *snapshot* baru yang merupakan hasil dari *three way merge* dan secara otomatis akan membuat *commit* yang baru seperti yang terlihat pada Gambar 13. Hal ini disebut sebagai *merge commit* karena memiliki lebih dari 2 *parent*.



Gambar 12: 3 *snapshot* yang digunakan dalam *three way merge*

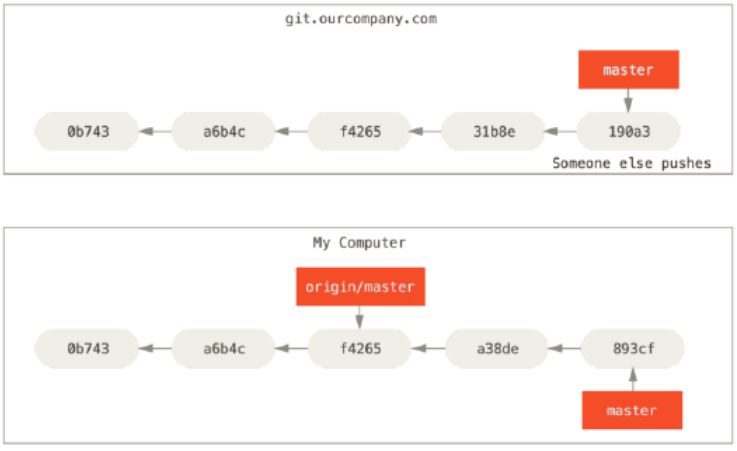


Gambar 13: *Merge commit*

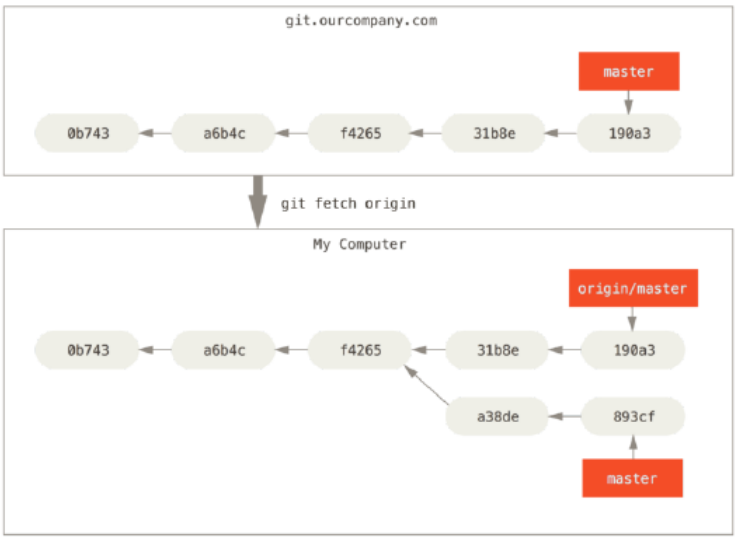
Merge pada *Git* mungkin akan menimbulkan konflik. Hal ini dapat terjadi apabila *file* yang sama pada kedua *branch* tersebut diubah pada bagian yang sama. Ketika mengetikan perintah *git merge*, maka *Git* tidak akan membuat *merge commit* secara otomatis. Proses *merge* akan dijeda sesudah konflik tersebut sudah diselesaikan. Untuk menangani konflik tersebut, pilihlah salah satu *branch*. Maksdud dari memilih salah satu *branch* adalah dengan mengubah *file* yang berada pada salah satu *branch*. Sesudah mengubah *file* pada branch yang dipilih, maka *Git* akan *merge branch* jika tidak ada konflik lagi.

Remote Branches

Remote-tracking branches adalah referensi dari *state remote branches*. Referensi tersebut merupakan referensi lokal yang hanya dapat dipindahkan oleh *Git* untuk memastikan jika referensi tersebut merepresentasikan *state* dari *remote repository*. <*remote*>/<*branches*> merupakan *remote-tracking branches*. Jika ingin mengecek *file* pada *branch master* yang berada dalam *remote origin*, maka pengguna harus mengecek *branch origin/master*. Sama seperti *branch master*, *origin* juga merupakan penamaan *remote* secara otomatis ketika *clone repository*. Jika pengguna mengubah *branch* lokal maka *branch* milik server tidak akan berubah dan hanya *pointer* pada lokal saja yang berubah. Maka dari itu *branch* di lokal dan *branch* di *server* bisa saja berbeda seperti yang terlihat pada Gambar 14. Untuk mensinkron *branch* di lokal dan *branch* di *server*, gunakan peirntah *git fetch* diikuti dengan nama *remote*. Dengan cara ini, beberapa data yang belum dimiliki akan diambil dari *server*, meng-*update* basis data lokal dan memindahkan *pointer* ke posisi yang terbaru seperti yang terlihat pada Gambar 15.



Gambar 14: Perbedaan pada *branch* lokal dan *remote*



Gambar 15: *Update remote-tracking branches* menggunakan perintah *git fetch*

Jika ingin membagikan *branch* ke pengguna lain, pengguna harus *push branch* tersebut ke *remote* karena *branch* lokal tidak sinkron secara otomatis dengan *remote*. Perintah yang digunakan untuk *push* adalah *git push* diikuti dengan nama *remote* dan nama *branch*.

Check out branch lokal dari *remote-tracking branch* secara otomatis akan membuat *tracking branch*. *Tracking branch* adalah *branch* lokal yang memiliki hubungan langsung dengan *branch remote*. Jika berada pada *tracking branch* dan mengetikan perintah *git pull*, secara otomatis *Git* mengetahui *server* mana yang akan di-*fetch* dan *branch* apa yang akan di-*merge*. Bila *clone repository*, maka secara otomatis akan membuat sebuah *branch* yang bernama master yang men-*track origin/master*. Untuk mengatur *tracking branch*, perintah yang digunakan adalah *git checkout -b <branch> <remote>/<branch>*. *Git* menyediakan perintah *git checkout –track <remote>/<branch>* sebagai shortcut dari perintah checkout sebelumnya. Perintah *git checkout* juga dapat digunakan untuk mengatur *branch* lokal dengan nama yang berbeda dari *branch remote*. Jika sudah memiliki *branch* lokal dan ingin mengatur *branch* tersebut ke *branch remote* yang sudah di-*pull*, gunakan opsi *-u* atau *–set-upstream-to* pada perintah *git branch*. Untuk melihat *tracking branch* yang sudah diatur, gunakan opsi *-vv* pada perintah *git branch*. Perintah ini akan menampilkan *list* dari *branch* lokal dengan informasi tambahan mengenai *tracking* pada setiap *branch* dan apakah *branch* lokal tersebut memiliki *ahead, behind* atau keduanya. *Ahead* adalah ada *commit* lokal yang belum di-*push* ke *server*, sedangkan *behind* adalah *commit* yang belum digabungkan. Perintah ini tidak langsung mengambil datanya dari *server* tetapi data tersebut merupakan data saat terakhir *fetch* dari *server*. Untuk mendapatkan data yang terbaru, harus *fetch* dari semua *remote* kemudian mengetikan perintah *git branch -vv*.

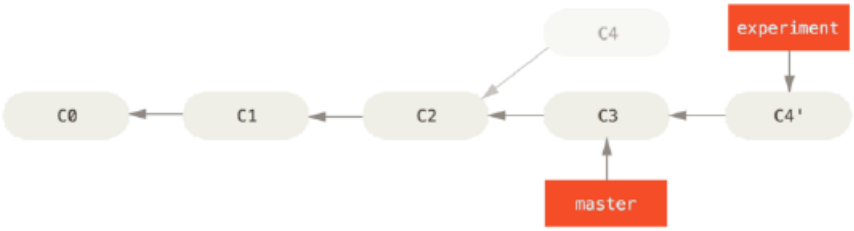
Perintah git fetch akan mengambil semua perubahan yang ada pada *server* yang tidak dimiliki oleh *branch lokal*, tetapi tidak mengubah *working directory* yang sesuai dengan *branch remote*. Perintah *git pull* digunakan untuk mengubah *working directory*. Perintah ini akan melihat *server* dan *branch* yang sedang di-*track*, mengambil data dari *server* tersebut dan menggabungkanya. Singkatnya, perintah *git pull* merupakan gabungan dari perintah *git fetch* dengan *git merge*.

Branch pada *remote* dapat dihapus dengan menggunakan opsi *–delete* pada perintah *git push*. *Branch* pada *remote* tidak sepenuhnya dihapus, tetapi hanya *pointer*nya saja yang dihilangkan.

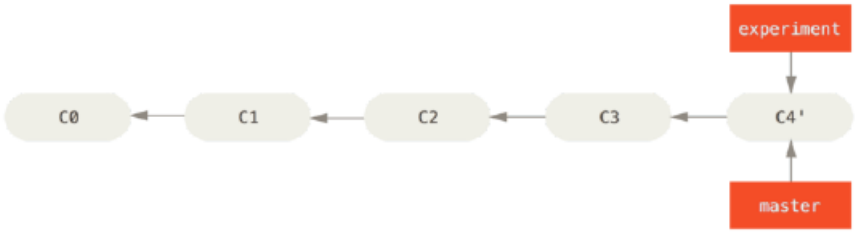
Jika *branch* tidak sengaja terhapus, maka data pada branch dapat dikembalikan/di*back-up*.

Rebasing

Selain *merge*, ada cara lain untuk menggabungkan kedua *branch* yaitu *rebasing*. Cara kerja dari *rebasing* adalah mencari *ancestor* yang sama dari kedua *branch*, mendapatkan perbedaan setiap *commit* pada *branch* saat ini, menyimpan perbedaan tersebut pada *file* sementara, mengatur ulang *branch* ke *commit* yang sama dengan *branch* yang akan di*rebase*, dan menerapkan setiap perubahannya. Contoh *rebasing* dapat dilihat pada Gambar 16. *Commit* C4 pada *branch experiment* berpindah dari C4 ke C4' yang berada di atas C3. Setelah *rebasing*, *merge* kedua *branch* tersebut sehingga hasilnya terlihat seperti pada Gambar 17. Untuk *rebasing*, gunakan perintah *git rebase* kemudian diikuti nama *branch* yang ingin di*rebase*.



Gambar 16: *Rebasing commit* C4 ke C3



Gambar 17: *Merge branch* setelah *rebasing*

Hasil terakhirnya tidak berbeda dengan menggunakan perintah *merge*, namun *rebasing* membuat histori menjadi lebih sedikit dibandingkan dengan *merge*. *Rebasing* juga berguna dalam berkontribusi pada proyek yang bukan milik sendiri. Hal ini akan mempermudah kerja pemilik proyek, karena pemilik proyek hanya tinggal *clean apply* saja.

GitHub

GitHub merupakan *single host* terbesar untuk *Git repository* dan sebagai titik tengah dari kolaborasi untuk jutaan pengembang dan proyek. Persentase terbesar dari semua *Git repository* di*hosting* di *GitHub* dan banyak proyek *open-source* menggunakanya untuk *Git hosting, code review, issue tracking* dan lainnya.

Fork

Jika pengguna ingin berkontribusi pada proyek yang sudah ada dan pengguna tidak memiliki akses untuk *push*, maka pengguna dapat *fork* proyek tersebut. Ketika proyek tersebut telah di-*fork*, *GitHub* akan membuatkan sebuah *copy/clone* dari proyek tersebut yang sekarang sudah menjadi milik penggunanya dan dapat di*push*. Orang lain dapat *fork* proyek, *push* proyek, dan berkontribusi dalam perubahan tersebut dan menyarankan untuk menggabungkan perubahan tersebut

dengan *repository* aslinya dengan membuat *Pull Request*.

Berikut adalah langkah-langkah untuk berkolaborasi dalam GitHub:

(a) *Fork* proyek yang diinginkan.
(b) Buat topik *branch* dari *master*.
(c) Lakukan *commit* untuk memperbaiki proyek.
(d) *Push branch* ke proyek *GitHub*.
(e) Buka *Pull Request* di *GitHub*.
(f) Diskusikan dan *commit* proyek tersebut apabila proyek tersebut masih membutuhkan perbaikan.
(g) Pemilik proyek *merges*/menggabungkan atau menutup *Pull Request*.

Pull Request
Pull Request membuka tempat diskusi untuk *owner(pemilik repository)* dan kontributor sehingga dapat berkomunikasi tentang perubahan tersebut sampai *owner* merasa puas dan senang. Setelah itu *owner* akan *merge*/menggabungkan perubahan tersebut. Setelah kontributor sudah membuat *Pull Request*, pemilik proyek dapat melihat saran perubahan proyek dari orang lain dan memberikan komentar/keterangan pada perubahan tersebut. Pemilik proyek dapat melihat perbedaan pada kode pemilik proyek dengan perubahan yang disarankan tersebut dan pemilik proyek dapat mengomentari baris pada kode tersebut. Orang lain dapat memberikan komentar pada *Pull Request*. Sesudah pemilik proyek memberikan keterangan tentang perubahan tersebut, kontributor menjadi tahu apa yang harus dilakukan agar perubahan tersebut dapat disetujui. Apabila perubahan tersebut membuat pemilik proyek puas, pemilik proyek akan *merge* perubahan tersebut dengan proyek aslinya dan otomatis akan menutup *Pull Request*.

2. **Melakukan analisis dan menentukan objek-objek pada Snake 360**
Status : Ada sejak rencana kerja skripsi.
Hasil :

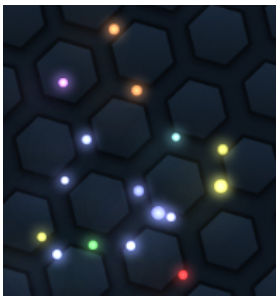
- **Analisis Permainan *Snake* yang Sudah Ada**
Permainan *Snake* yang akan dianalisis adalah *Slither.io*. *Slither.io* adalah permainan *web* yang dapat dimainkan oleh lebih dari 1 pemain(*multiplayer*). Cara bermainya mirip seperti permainan *Snake* pada umumnya yaitu ular harus memakan makanan untuk mendapatkan skor. Dalam permainan ini, setiap pemain berkompetisi untuk menjadi pemain terbaik dengan cara mendapatkan skor sebanyak-banyaknya. Pemain akan kalah apabila ular milik pemain menabrak ular milik pemain lain.

Ular dan Makanan
Ular pada *Slither.io* dibentuk dengan menggunakan sekumpulan lingkaran yang saling berdempetan satu sama lain seperti pada Gambar 18. Bagian kepala pada ular ditandai menggunakan sepasang mata. Ketika memakan makanan, tubuh ular akan memanjang dengan menambahkan sebuah lingkaran pada bagian ekor ular. Setiap memulai permainan, tubuh ular akan memiliki warna yang ditentukan secara acak.

Makanan pada *Slither.io* berbentuk lingkaran. Makanan ini ada yang berukuran besar dan ada yang berukuran kecil. Makanan ini tersebar pada labirin, jumlahnya sangat banyak dan warnanya bermacam-macam. Gambar 19 merupakan sekumpulan makanan yang terdapat pada labirin. Setiap makanan akan menambah skor sebanyak 1 poin.

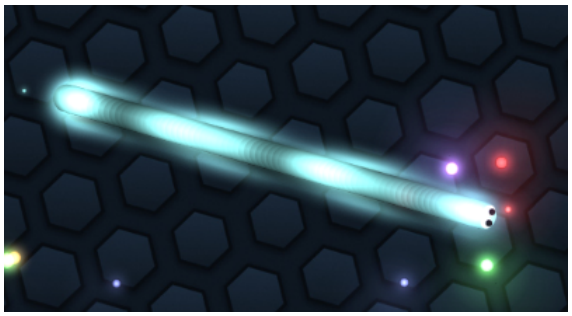
Gambar 18: Ular pada *Silther.io*

Gambar 19: Makanan pada *Slither.io*



Pergerakan Ular

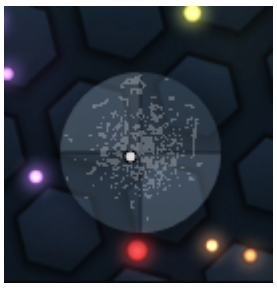
Ular pada *Slither.io* digerakan dengan menggunakan keyboard dan *mouse*. Tombol ke kiri akan membuat ular bergerak berlawanan arah jarum jam dan tombol ke kanan akan membuat ular bergerak searah jarum jam. Semakin lama tombol ditekan, maka ular akan berbelok lebih cepat. Kursor pada *mouse* membuat ular bergerak ke arah posisi kursor tersebut. Ular dapat melaju dengan cepat(*speed up*) dengan menekan tombol *mouse* kiri seperti yang terdapat pada Gambar 20. Ketika ular sedang melaju dengan cepat, total skor yang didapat akan berkurang.



Gambar 20: Ular sedang melaju dengan cepat(*speed up*)

Labirin

Labirin pada *Slither.io* hanya ada 1 saja. Labirin ini berbentuk lingkaran yang sisinya dikelilingi oleh dinding. Apabila ular menabrak dinding labirin, maka permainan akan berakhir. Labirin ini cukup besar sehingga sangat kecil kemungkinan ular untuk menabrak dinding labirin. Gambar 21 menunjukan peta labirin pada *Slither.io*. Pada peta labirin tersebut terdapat sekumpulan titik bewarna abu-abu yang merepresentasikan makanan.



Gambar 21: Peta labirin pada *Slither.io*

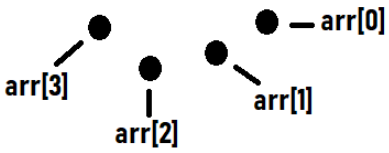
- **Analisis Sistem yang Dibangun**

  Permainan *Snake* 360 yang akan dibangun memiliki cara bermain yang mirip seperti permainan Snake pada umumnya. Perbedaan antara *Snake* 360 dengan permainan *Snake* pada umumnya adalah *Snake* 360 dapat menambahkan level dan labirin sendiri.

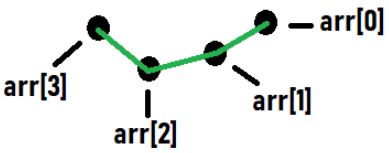
Menggambar Ular dan Apel

Tubuh ular dibuat menggunakan sekumpulan *line*/garis pendek. Setiap bagian tubuh ular memiliki panjang sebesar 1 *pixel* dan lebar tubuhnya sebesar 5 *pixel*. Bagian tubuh ular dibuat pendek untuk memudahkan pengecekan jika terjadi ular menabrak tubuhnya sendiri. Untuk lebar ular, disesuaikan dengan besar apel yaitu 10 *pixel*. Setiap bagian tubuh ular memiliki koordinat masing-masing. Koordinat setiap bagian tubuh disimpan pada sebuah *array* agar menggambar ular menjadi lebih mudah. Dalam tahap ini, tubuh ular masih berupa sekumpulan titik-titik yang merupakan koordinat bagian tubuh ular seperti pada Gambar 22. Algoritma untuk menggambar ular adalah dengan mengambil koordinat bagian tubuh ular mulai dari elemen *array* paling pertama(arr[0]) dan elemen *array* selanjutnya(arr[1]) lalu buat garis yang *start point*nya adalah elemen pertama(arr[0]) dan *end point*nya adalah elemen *array* kedua(arr[1]). Setelah itu ambil koordinat elemen *array* yang merupakan *end point* pada garis sebelumnya(arr[1]) dengan elemen *array* selanjutnya(arr[2]) dan gambar garisnya. Lakukan hal tersebut sampai *end point* garis mencapai elemen *array* paling akhir. Setelah digambar maka ular akan terlihat seperti Gambar 23.

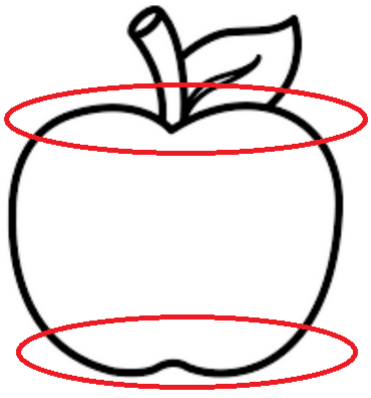
Gambar 22: Koordinat bagian tubuh ular pada *array*



Gambar 23: Tubuh ular setelah digambar menggunakan garis



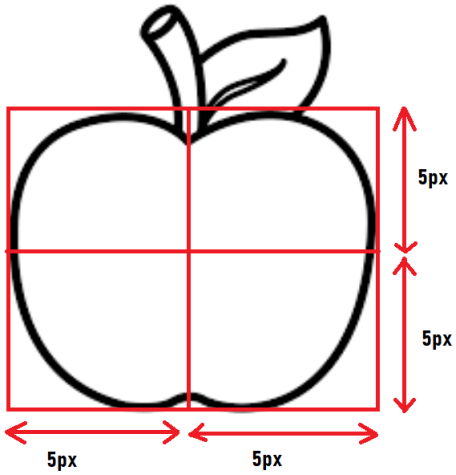
Untuk membuat apel digunakan *quadratic Bézier curve*. Kurva ini digunakan untuk membuat bagian-bagian apel yang melengkung. Bagian tersebut ditandai dengan lingkaran bewarna merah seperti yang ditunjukan pada Gambar 24(gambar diambil dari pinterest:https://www.pinterest.com/pin/690317449105509454)



Gambar 24: Bagian pada apel(lingkaran merah) yang akan dibuat menggunakan kurva

Pertama, tentukan besar apel yang ingin dibuat. Dalam permainan ini besar apel yang dibuat adalah 10 pixel. Besar apel dibuat lebih besar dari lebar ular karena jika besar apel sama dengan lebar ular, besar apel terlihat sangat kecil. Selain itu, apel ini digambar pada layout yang berbentuk persegi. Layout persegi ini juga dapat mempermudah penggambaran apel. Karena menggunakan layout persegi, maka origin terletak pada titik sudut di sebelah kiri atas. Setelah

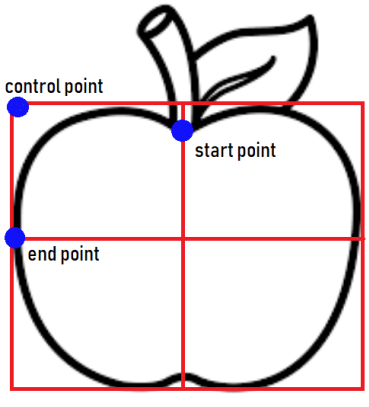
itu, gambar setiap bagian apel. Bagian apel dibagi menjadi 4 seperti pada Gambar 25 sehingga besar setiap bagian apel tersebut adalah 5 pixel.



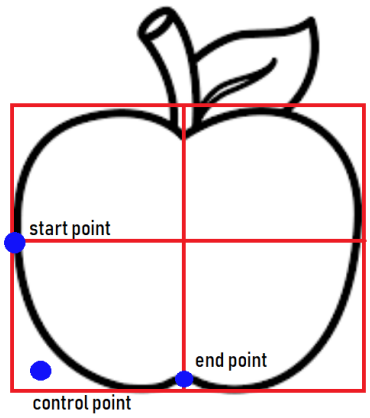
Gambar 25: Pembagian gambar apel dengan layout persegi beserta ukuran pada setiap bagian

Gambar bagian atas apel terlebih dahulu. Gunakan method moveTo() untuk menentukan titik mulainya. Titik mulainya terletak pada bagian tengah atas apel yang melengkung ke dalam. Dari titik itu, buat kurva yang control pointnya adalah titik ujung layout persegi. Jika ingin menggambar bagian kiri apel terlebih dahulu maka control pointnya adalah titik ujung kiri layout tersebut. Setelah itu, tentukan end point kurva tersebut. Pada Gambar 26 terdapat start point, control point dan end point untuk membuat bagian sisi kiri atas apel. Sesudah itu, buatlah bagian bawah apel. Caranya sama seperti sebelumnya namun control pointnya dan end pointnya berbeda. Posisi control pointnya sedikit menjorok ke dalam dan posisi end pointnya terdapat di tengah bawah seperti pada Gambar 27. Start point tidak perlu diatur lagi, karena start pointnya sudah tergantikan dengan posisi end point pada kurva sebelumnya. Sampai pada bagian ini, bagian kiri apel sudah selesai dibuat. Untuk membuat bagian kanan apel, caranya sama seperti membuat bagian kiri apel. Karena bagian kiri apel simetris dengan bagian kanan apel, maka hanya perlu mengubah control point dan end pointnya saja. Dengan memanfaatkan bentuk simetris dari apel, maka jarak antara control point dan end point pada bagian kiri apel dengan batasan tengah sama dengan jarak antara control point dan end point dengan batas tengah pada bagian kanan apel.

Gambar 26: Start point, control point dan end point untuk menggambar apel bagian kiri atas

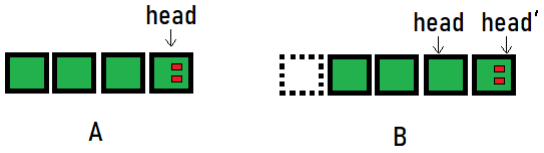


Gambar 27: Start point, control point dan end point untuk menggambar apel bagian kiri bawah



Pergerakan Ular

Untuk membuat ular bergerak maju, dilakukan penambahan kepala dan pembuangan ekor secara bersamaan ketika ular sedang bergerak maju. Ilustrasinya dapat dilihat pada Gambar 28. Untuk membuat ular bergerak dengan menggunakan cara pada Gambar 28, algoritmanya adalah sebagai berikut : Pertama, semua elemen array akan dishift/digeser dan elemen pertama akan digantikan dengan koordinat yang baru. Setelah itu dilakukan pengecekan apakah panjang tubuh ular lebih besar dari jumlah elemen array tubuh ular. Jika benar, maka tidak dilakukan pembuangan elemen terakhir dan jika salah, maka tidak akan dilakukan apa-apa.



Gambar 28: Ilustrasi ular sebelum bergerak maju(A) dan setelah bergerak maju(B)

Kecepatan ular pada permainan ini adalah 1 sampai 5 *pixel per frame*. Kecepatan maksimal ular tidak boleh melebihi lebar tubuh ular. Jika kecepatanya melebihi lebar ular, maka ketika terjadi tabrakan dengan tubuhnya sendiri, kepala ular tidak akan bertabrakan dengan tubuhnya. Kepala ular akan terlihat seolah-olah melompati tubuhnya sendiri. Dalam permainan ini, kecepatan ular adalah 2 *pixel per frame*, karena dengan kecepatan 1 *pixel per frame*, ular terlihat bergerak lebih lambat.

Ular dapat berbelok dengan menggunakan tombol pada *keyboard*. Tombol ke kiri akan membuat ular bergerak melawan arah jarum jam dan tombol ke kanan akan membuat ular akan bergerak searah jarum jam. Pada permainan yang akan dibuat ini, digunakan sudut sebagai nilai untuk membuat ular dapat bergerak 360°. Jika menekan tombol ke kiri maka sudut akan berkurang dan jika menekan tombol ke kanan maka sudut akan bertambah. Ketika menambahkan dan mengurangi sudut, perlu dilakukan pengecekan apabila nilai sudut valid atau tidak. Karena nilai sudut yang valid adalah antara nilai 0 sampai 360, maka apabila nilai sudut kurang dari 0, ubahlah sudut tersebut menjadi 360 dan apabila nilai sudut lebih besar dari 360, ubahlah nilai sudut tersebut menjadi 0. Dibutuhkan rumus trigonometri untuk menentukan posisi kepala ular. Untuk menghitung posisi koordinat x, digunakan *sinus* sedangkan untuk menghitung posisi koordinat y menggunakan *cosinus*. Lalu koordinat x dan y pada kepala ular akan ditambahkan dengan hasil perhitungan *cosinus* dan *sinus*.

Labirin

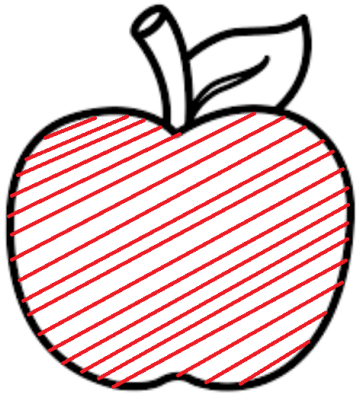
Dinding labirin akan dibuat dengan menggunakan garis. Cara pembuatanya sama dengan membuat tubuh ular yaitu dengan menggunakan titik-titik yang dihubungkan dengan garis yang pendek. Semua titik-titik tersebut akan disimpan pada sebuah array. Dinding labirin dapat juga dibuat dengan menggunakan kurva dikarenakan pergerakan ular yang sudah dapat bergerak 360°. Level pada labirin dapat ditentukan berdasarkan kerumitan labirin. Labirin yang memiliki dinding yang banyak dan kompleks akan mendapatkan level yang lebih tinggi dibandingkan dengan labirin yang memiliki sedikit dinding dan lebih simpel.

Pengecekan tabrakan(*Collision Detection*)

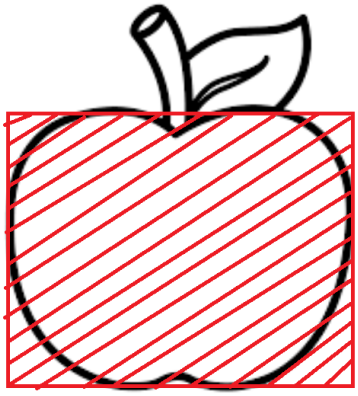
Pada permainan ini terdapat pengecekan tabrakan yang dapat mengecek apakah ular sudah me-

makan makanan, ular menabrak tubuhnya sendiri, dan ular menabrak dinding labirin. Seluruh pengecekan ini akan dilakukan pada setiap *frame*. Pada pengecekan tabrakan pada apel dan ular, hanya perlu mengecek tabrakan antara kepala ular dengan apel. Karena jalur yang dilalui oleh kepala ular, akan selalu dilalui oleh bagian tubuh ular. Dengan kata lain, bagian tubuh ular akan mengikuti ke mana kepala ular akan bergerak. Dengan ini, tidak perlu dilakukan *collision detection* antara bagian tubuh ular dengan apel. Cukup hanya dengan mengecek tabrakan antara kepala ular dengan apel saja. Untuk mengetahui terjadinya tabrakan antara ular dengan apel, maka akan dibuat daerah tabrakan pada apel. Daerah tabrakan ini digunakan untuk mengecek apakah 2 benda saling bertabrakan satu sama lain. Daerah tabrakan pada apel ditandai dengan arsiran bewarna merah yang terdapat pada Gambar 29. Namun, untuk membuat daerah tabrakan ini cukup sulit ketika mengecek adanya tabrakan antara ular dengan apel terutama pada bagian lengkungan pada apel. Karena itu, daerah tabrakan pada apel dibuat dengan menggunakan bentuk persegi seperti pada Gambar 30. Jika posisi kepala ular berada di dalam daerah tabrakan apel, maka dipastikan bahwa ular tersebut sudah memakan apel. Algoritma untuk mengecek tabrakan adalah sebagai berikut : cek apakah koordinat x dari kepala ular lebih besar dari posisi sisi kiri daerah tabrakan dan lebih kecil dari posisi sisi kanan daerah tabrakan. Kemudian cek apakah koordinat y dari kepala ular lebih besar dari posisi sisi atas daerah tabrakan dan lebih kecil dari posisi sisi bawah daerah tabrakan. Jika posisi kepala ular berada memenuhi ketentuan tersebut, maka kepala ular berada di dalam daerah tabrakan apel.

Gambar 29: Koordinat bagian tubuh ular pada *array*



Gambar 30: Tubuh ular setelah digambar menggunakan garis



Untuk mengecek tabrakan antara ular dengan tubuhnya sendiri adalah dengan mengecek tabrakan antara kepala ular dengan seluruh bagian tubuh ular. Algoritma pengecekanya adalah sebagai berikut : jika koordinat x kepala ular lebih kecil dari koordinat x bagian tubuh ular dikurangi panjang dari bagian tubuh ular dan lebih besar dari koordinat x bagian tubuh ular ditambah dengan panjang dari bagian tubuh ular. Kemudian dicek apabila koordinat y kepala ular lebih kecil dari koordinat y bagian tubuh ular dikurangi panjang dari bagian tubuh ular dan lebih besar dari koordinat y bagian tubuh ular ditambah dengan panjang dari bagian tubuh ular. Apabila posisi kepala ular memenuhi ketentuan tersebut, maka posisi kepala ular berada di dalam daerah tabrakan pada sebuah bagian tubuh ular.

Untuk mengecek tabrakan dengan labirin, algoritmanya sama dengan mengecek tabrakan antara ular dengan tubuh ular. Karena cara pembuatan labirin sama hampir sama dengan cara pembuatan tubuh ular, maka dilakukan pengecekan antara kepala ular dengan setiap dinding labirinya.

3. **Merancang algoritma untuk menggambar tubuh ular, pergerakan ular dan membuat labirin.**
   Status : Ada sejak rencana kerja skripsi.
   Hasil : Sudah tercakup pada poin 2

4. **Mengimplementasikan keseluruhan algoritma.**
   Status : Ada sejak rencana kerja skripsi.
   Hasil : Belum ada perkembangan.

5. **Menambahkan labirin menggunakan *pull request* pada *Github*.**
   Status : Ada sejak rencana kerja skripsi.
   Hasil : Belum ada perkembangan.

6. **Melakukan pengujian dan *debugging*.**
   Status : Ada sejak rencana kerja skripsi.
   Hasil : Belum ada perkembangan.

7. **Menulis dokumen skripsi**
   Status : Ada sejak rencana kerja skripsi.
   Hasil : Dokumen skripsi sudah ditulis sampai bab 3. Bab 1 berisikan latar belakang, rumusan masalah, tujuan, batasan masalah, metodologi, dan sistematika pembahasan dari penelitian yang dilakukan. Bab 2 berisikan dasar-dasar teori yang menunjang penelitian ini yaitu: pengertian *Snake*, HTML5 Canvas, *Javascript*, *jQuery*, dan *Git*. Bab 3 berisikan analisis sistem yang sudah ada, analisis sistem yang dibangun dan analisis berorientasi objek.

6 Pencapaian Rencana Kerja

Langkah-langkah kerja yang berhasil diselesaikan dalam Skripsi 1 ini adalah sebagai berikut:

1. Melakukan studi literatur tentang HTML5, *JavaScript*, *jQuery* dan *Git*.
2. Melakukan analisis dan menentukan objek-objek pada *Snake* 360.
3. Merancang algoritma untuk menggambar tubuh ular, pergerakan ular dan membuat labirin.
4. Menulis dokumen skripsi sampai bab 3.

7 Kendala yang Dihadapi

Kendala - kendala yang dihadapi selama mengerjakan skripsi :

- Terlalu banyak tugas baik individu maupun kelompok dari mata kuliah lain sehingga pengerjaan skripsi menjadi terhambat.
- Terlalu banyak godaan berupa hiburan terutama game.

Bandung, 19/11/2018

Evelyn Wijaya

Menyetujui,

Nama: Pascal Alfadian Nugroho
Pembimbing Tunggal